Red Hat Process Automation Manager 7.2

DRL ルールを使用したデシジョンサービスの作成

Last Updated: 2019-03-27

Red Hat Process Automation Manager 7.2 DRL ルールを使用したデシジョンサービスの作成

Red Hat Customer Content Services
brms-docs@redhat.com

法律上の通知

概要

本書は、Red Hat Process Automation Manager 7.2 で、DRL ルールを使用してデシジョンサービスを作成する方法を説明します。

目次

前書き

ビジネスルール開発者は、Business Central で DRL (Drools Rule Language) デザイナーを使用してビジネスルールを定義できます。DRL ルールは、Business Central におけるその他のルールアセットとは異なり、ガイド付きまたは表形式ではなく、フリーフォームの `.drl` テキストファイルで直接定義できます。このような DRL ファイルは、プロジェクトのデシジョンサービスの中心となります。

前提条件/事前作業

DRL ルールのチームおよびプロジェクトが Business Central に作成されていて、各アセットが、チームに割り当てられたプロジェクトに関連付けられています。詳細は**『デシジョンサービスの使用ガイド』** を参照してください。

第1章 RED HAT PROCESS AUTOMATION MANAGER におけるルール作成アセット

Red Hat Process Automation Manager は、デシジョンサービスにビジネスルールを作成するのに使用するアセットを提供します。ルール作成アセットはそれぞれ長所が異なるため、ゴールおよびニーズに適したアセットを 1 つ、または複数を組み合わせて使用できます。

デシジョンサービスでルールを作成する最適な方法を選択できるように、以下の表で、Business Central のルール作成アセットを紹介します。

表1.1 Business Central におけるルール作成アセット

アセット	主な特徴	ドキュメンテーション
ガイド付きデシジョンテーブル	• Business Central の UI ベースのテーブルデザイナーで作成するルールのテーブル • デシジョンテーブルのスプレッドシートをアップロードする代わりに、ウィザードを用いて作成する • 条件を満たした入力に、フィールドとオプションを提供する • ルールテンプレートを作成するテンプレートキーと値をサポートする • その他のアセットではサポートされていない、ヒットポリシー、リアルタイム検証などの追加機能をサポートする • コンパイルエラーを最小限に抑えるため、制限されているテーブル形式でルールを作成するのに最適	ガイド付きデシジョンテーブルを使用したデシジョンサービスの作成
アップロードしたデシジョンテーブル	• Business Central にアップロードした XLS 形式または XLSX 形式のデシジョンテーブルスプレッドシート • ルールテンプレートを作成するテンプレートキーと値をサポートする • Business Central 外で管理しているデシジョンテーブルでルールを作成するのに最適 • アップロード時に適切にルールをコンパイルするために厳しい構文要件がある	アップロードしたデシジョンテーブルを使用したデシジョンサービスの作成

アセット	主な特徴	ドキュメンテーション
ガイド付きルール	• Business Central の UI ベースのルールデザイナーで作成する個々のルール • 条件を満たした入力に、フィールドとオプションを提供する • コンパイルエラーを最小限に抑えるため、制御されている形式で単独のルールを作成するのに最適	**ガイド付きルールを使用したデシジョンサービスの作成**
ガイド付きルールテンプレート	• Business Central の UI ベースのテンプレートデザイナーで作成する再利用可能なルール構造 • 条件を満たした入力に、フィールドとオプションを提供する • (このアセットの基本となる) ルールテンプレートを作成するテンプレートキーと値をサポートする • ルール構造が同じで、定義したフィールド値が異なるルールを多数作成するのに最適	**ガイド付きルールテンプレートを使用したデシジョンサービスの作成**
DRL ルール	• `.drl` テキストファイルに直接定義する個々のルール • 最も柔軟性が高く、ルールと、ルール動作に関するその他の技術を定義できる • スタンドアロン環境で作成し、 Red Hat Process Automation Manager に統合可能 • 高度な DRL オプションが必要なルールを作成するのに最適 • ルールを適切にコンパイルするために厳密な構文要件がある	**DRL ルールを使用したデシジョンサービスの作成**

第2章 DRL (DROOLS RULE LANGUAGE) ルール言語

DRL (Drools Rule Language) ルールは、**.drl** テキストファイルに直接定義するビジネスルールです。このような DRL ファイルは、Business Central の他のすべてのルールアセットが最終的にレンダリングされるソースとなります。DRL ファイルを Business Central インターフェースで作成および管理したり、外部の Red Hat Developer Studio、Java オブジェクト、Maven アーキタイプを使用して作成したりできます。DRL ファイルには、ルールの条件 (**when**) およびアクション (**then**) を定義するルールを最低でも 1 つ以上追加できます。Business Central の DRL デザイナーでは、Java、DRL、および XML の構文が強調表示されます。

DRL ルールに関連するデータオブジェクトはすべて、DRL ルールと同じ Business Central プロジェクトパッケージに置く必要があります。同じパッケージのアセットはデフォルトでインポートされます。その他のパッケージの既存アセットは、DRL ルールを使用してインポートできます。

第3章 データオブジェクト

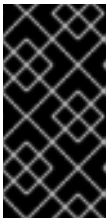
データオブジェクトは、作成するルールアセットの構成要素です。データオブジェクトは、プロジェクトで指定したパッケージに Java オブジェクトして実装されているカスタムのデータ型です。たとえば、データフィールド **Name**、**Address**、および **Date of Birth** を使用して **Person** オブジェクトを作成し、ローン申し込みルールに詳細な個人情報を指定できます。このカスタムのデータ型は、アセットとディシジョンサービスがどのデータに基づいているかを指定します。

3.1. データオブジェクトの作成

以下の手順は、データオブジェクトを作成する際の一般的な概要で、特定のビジネスプロセスに固有のものではありません。

手順

1. Business Central で、**Menu** → **Design** → **Projects** に移動して、プロジェクト名をクリックします。
2. **Add Asset** → **Data Object** をクリックします。
3. 一意の **データオブジェクト** 名を入力し、**パッケージ** を選択します。これにより、その他のルールアセットでもデータオブジェクトを利用できるようになります。同じパッケージに、同じ名前のデータオブジェクトを複数作成することはできません。指定の DRL ファイルで、どのパッケージからでもデータオブジェクトをインポートできます。

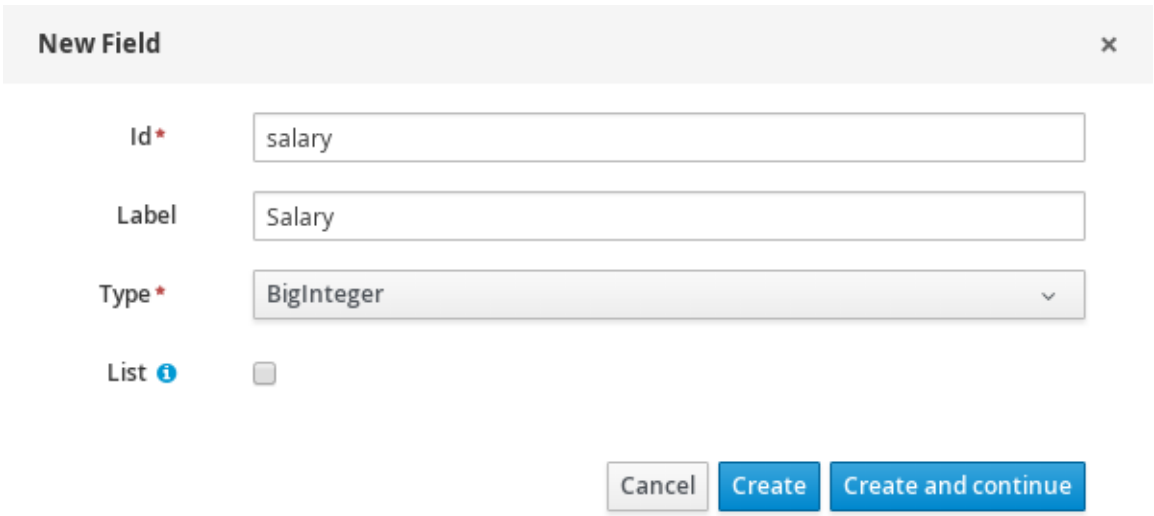


別のパッケージからのデータオブジェクトのインポート

別のパッケージから直接アセットデザイナーに、既存のデータオブジェクトをインポートすることができます。プロジェクトで関連するルールアセットを選択し、アセットデザイナーで **Data Objects** → **New item** に移動して、インポートするオブジェクトを選択します。

4. データオブジェクトを永続化するには、**Persistable** チェックボックスを選択します。永続型データオブジェクトは、JPA 仕様に準じてデータベースに保存できます。デフォルトの JPA は Hibernate です。
5. **OK** をクリックします。
6. データオブジェクトデザイナーで **add field** をクリックして、**Id** 属性、**Label** 属性、**Type** 属性を使用するオブジェクトにフィールドを追加します。必須属性にはアスタリスク (*) マークが付いています。
   - **Id:** フィールドの一意の ID を入力します。
   - **Label:** (任意) フィールドのラベルを入力します。
   - **Type:** フィールドのデータ型を入力します。
   - **List:** このチェックボックスを選択すると、このフィールドで、指定したタイプのアイテムを複数保持できるようになります。

図3.1 データオブジェクトへのデータフィールドの追加



7. **Create** をクリックして、新しいフィールドを追加します。**Create and continue** をクリックすると、新しいフィールドが追加され、別のフィールドを引き続き作成できます。

注記

フィールドを編集するには、フィールド行を選択し、画面右側の **general properties** を使用します。

第4章 BUSINESS CENTRAL における DRL ルールの作成

Business Central で、プロジェクトに対して DRL ルールを作成して管理できます。パッケージに作成またはインポートするデータオブジェクトに基づいて、各 DRL ファイルで、ルールの条件、アクション、そしてルールに関連するその他のコンポーネントを定義します。

手順

1. Business Central で、**Menu** → **Design** → **Projects** に移動して、プロジェクト名をクリックします。
2. **Add Asset** → **DRL ファイル** をクリックします。
3. 参考となる **DRL ファイル** 名を入力し、適切な **パッケージ** を選択します。指定するパッケージは、必要なデータオブジェクトが割り当てられている、またはこれから割り当てるパッケージにする必要があります。
   ドメイン固有言語 (DSL) アセットがプロジェクトに定義されている場合は、**Show declared DSL sentences** を選択することもできます。この DSL アセットは、DRL デザイナーで定義する条件およびアクションに使用できるオブジェクトです。
4. **OK** をクリックして、ルールアセットを作成します。
   新しい DRL ファイルが、**Project Explorer** の **DRL** パネルに追加されます。**Show declared DSL sentences** オプションを選択した場合は、**DSLR** パネルに追加されます。この DRL ファイルを割り当てたパッケージは、ファイルの上位にリストされます。
5. DRL デザイナーの左パネルの **Fact types** リストで、ルールに必要なすべてのデータオブジェクトとデータオブジェクトフィールドがリストされていることを確認します (それぞれを展開します)。リストされていない場合は、DRL ファイルの `import` 命令文を使用して、その他のパッケージから関連するデータオブジェクトをインポートするか、パッケージにデータオブジェクトを作成します。
6. データオブジェクトをすべて配置したら、DRL デザイナーの **Model** タブに戻り、以下のいずれかのコンポーネントで DRL ファイルを定義します。

   DRL ファイルのコンポーネント

   ```
   package  //automatic

   import

   function  //optional

   query  //optional

   declare   //optional

   rule

   rule

   ...
   ```

   - **package**: (自動) これは、DRL ファイルを作成し、パッケージを選択すると定義されます。

- **import**: このパッケージ、または DRL ファイルで使用するその他のパッケージのデータオブジェクトを指定します。パッケージとデータオブジェクトを **package.name.object.name** 形式で指定し、1 行につき 1 つインポートします。

  データオブジェクトのインポート

  ```
  import mortgages.mortgages.LoanApplication;
  ```

- **function**: (任意) DRL ファイルのルールが使用する関数を指定します。関数は、ルールのソースファイルにセマンティックコードを追加します。関数は、特に、ルールのアクション (**then**) 部分が繰り返し使用され、パラメーターだけがルールごとに異なる場合に便利です。DRL ファイルのルールで、関数を宣言したり、静的メソッドを関数としてインポートしたりして、ルールの アクション (**then**) 部分に、名前を指定して関数を使用します。

  ルールに関数を宣言して使用 (オプション 1)

  ```
  function String hello(String applicantName) {
      return "Hello " + applicantName + "!";
  }

  rule "Using a function"
    when
      eval( true )
    then
      System.out.println( hello( "James" ) );
  end
  ```

  ルールに関数をインポートして使用 (オプション 2)

  ```
  import function my.package.applicant.hello;

  rule "Using a function"
    when
      eval( true )
    then
      System.out.println( hello( "James" ) );
  end
  ```

- **query**: (任意) DRL ファイルのルールに関連するファクトに対してプロセスエンジンを検索するのに使用します。クエリーは、定義した条件セットを検索するため、**when** または **then** を指定する必要はありません。クエリー名は KIE ベースでグローバルとなるため、プロジェクトにあるその他のすべてのルールクエリーと重複しないようにする必要があります。クエリーの結果に戻るには、**ksession.getQueryResults("name")** を使用して、従来の **QueryResults** 定義を構成します (**"name"** はクエリー名)。これにより、クエリーの結果が返り、クエリーに一致したオブジェクトを取得できるようになります。DRL ファイルのルールに、クエリーと、クエリー結果パラメーターを定義します。

  ルールで、年齢が 21 歳未満の場合のクエリーと、その結果

  ```
  query "people under the age of 21"
      person : Person( age < 21 )
  end

  QueryResults results = ksession.getQueryResults( "people under
  ```

```
the age of 21" );
System.out.println( "we have " + results.size() + " people under
the age  of 21" );

rule "Underage"
  when
    application : LoanApplication( )
    Applicant( age < 21 )
  then
    application.setApproved( false );
    application.setExplanation( "Underage" );
end
```

- **declare**: (任意) DRL ファイルのルールが使用する新しいファクトタイプを宣言します。Red Hat Process Automation Manager の **java.lang** パッケージのデフォルトは **Object** ですが、必要に応じて DRL ファイルに別のタイプを宣言することもできます。DRL ファイルにファクトタイプを宣言すると、Java などの低級言語でモデルを作成せず、プロセスエンジンに直接新しいファクトモデルを定義するようになります。

  新しいファクトタイプの宣言および使用

```
declare Person
  name : String
  dateOfBirth : java.util.Date
  address : Address
end

rule "Using a declared type"
  when
    $p : Person( name == "James" )
  then   // Insert Mark, who is a customer of James.
    Person mark = new Person();
    mark.setName( "Mark" );
    insert( mark );
end
```

- **rule**: DRL ファイルで各ルールを定義します。ルールは、**rule "name"** 形式のルール名、ルールの動作 (**salience**、**no-loop** など) を定義する任意の属性、**when** および **then** 定義が続きます。同じパッケージにルール名を重複させることはできません。ルールの **when** 部分には、アクションを実行する条件が含まれます。たとえば、銀行が、ローンの申し込みを 21 歳以上に限定した場合、**Underage** ルールの **when** 条件は **Applicant(age < 21)** になります。ルールの **then** 部分には、ルールの条件が一致したときに実行するアクションが含まれます。たとえば、ローンの申込者が 21 歳に満たない場合、**then** アクションは **setApproved(false)** になり、申込者の年齢条件を満たしていないためにローンの申し込みは承認されません。条件 (**when**) およびアクション (**then**) は、パッケージで利用可能なデータオブジェクトに基づいて、任意の制約、バインディングなどのサポートされる DRL 要素を持つ、定められている一連のファクトパターンで構成されます。このパターンは、定義したオブジェクトにルールがどのように影響するかを指定します。

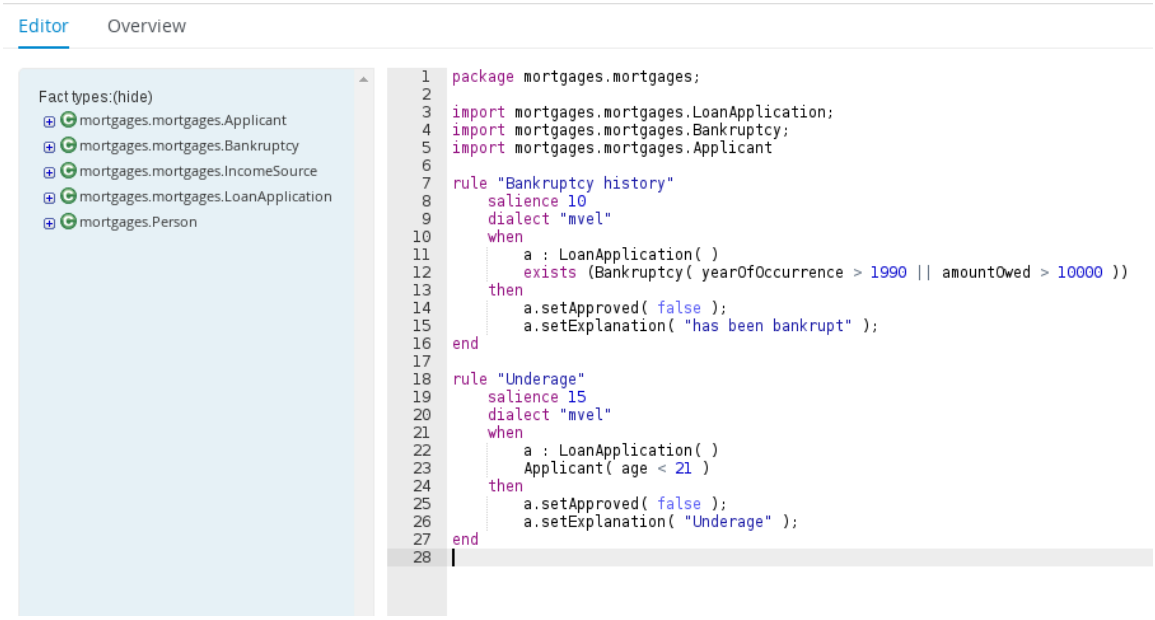
  申込者の年齢制限に関するルール

```
rule "Underage"
  salience 15
  dialect "mvel"
```

```
  when
    application : LoanApplication( )
    Applicant( age < 21 )
  then
    application.setApproved( false );
    application.setExplanation( "Underage" );
end
```

少なくても、各 DRL ファイルに **package** コンポーネント、**import** コンポーネント、**rule** コンポーネントを指定する必要があります。他のすべてのコンポーネントは任意です。

図4.1 必要なコンポーネントおよび任意のルール属性を持つ DRL ファイルのサンプル



7. ルールのコンポーネントをすべて定義したら、DRL デザイナーの右上ツールバーで **Validate** をクリックし、DRL ファイルの妥当性を確認します。ファイルの妥当性確認に失敗したら、エラーメッセージに記載された問題に対応し、DRL ファイルの構文およびコンポーネントをすべて見直し、エラーが表示されなくなるまで再度、ファイルを検証します。

8. DRL デザイナーで **Save** をクリックして、設定した内容を保存します。

DRL ルールに条件を追加する方法は「DRL ルールへの WHEN 条件の追加」を参照してください。

DRL ルールにアクションを追加する方法は「DRL ルールへの THEN アクションの追加」を参照してください。

4.1. DRL ルールへの WHEN 条件の追加

ルールの **when** 部分には、アクションを実行するのに必要な条件が含まれます。たとえば、銀行のローン申し込みに年齢制限 (21 歳以上) が必要な場合、**Underage** ルールの **when** 条件は **`Applicant( age < 21 )`** となります。パッケージで利用可能なデータオブジェクトに基づいて、指定した一連のパターンおよび制約と、任意のバインディング、その他のサポートされる DRL 要素で構成されます。

前提条件

- **package** は DRL ファイルに定義されます。これは、ファイルの作成時に行われます。

- ルールで使用したデータオブジェクトの **import** リストが、DRL ファイルの **package** 行の下に定義されます。データオブジェクトは、このパッケージ、または別の Business Central パッケージから使用できます。
- **rule** 名は、**package**、**import**、または DRL ファイル全体に適用されるその他の行の下に、**rule "name"** という形式で定義されます。同じパッケージでルール名を重複させることはできません。ルールの動作 (**salience**、**no-loop** など) を定義する任意のルール属性は、ルール名の下、**when** セクションの前に定義します。

手順

1. DRL デザイナーで、ルールに **when** を入力して、条件命令文を追加します。**when** セクションは、ルールの条件を定義するファクトパターンで構成されますが、ファクトパターンが 1 つも追加されない場合もあります。
when セクションを空にすると、プロセスエンジンで **fireAllRules()** を呼び出すたびに、**then** セクションのアクションが実行します。これは、プロセスエンジンのステートを設定するルールを使用する場合に便利です。

   条件のないルール

```
rule "bootstrap"
  when   // empty

  then   // actions to be executed once
    insert( new Applicant() );
end

// The above rule is internally rewritten as:

rule "bootstrap"
  when
    eval( true )
  then
    insert( new Applicant() );
end
```

2. 一致させる最初の条件のパターンを入力し、任意で制約、バインディング、およびサポートされる DRL 要素を入力します。基本的なパターンフォーマットは **patternBinding : patternType (constraints)** です。パターンは、パッケージで利用可能なデータオブジェクトに基づいており、**then** セクションのアクションを発生させるのに必要な条件を定義します。

   - **単純なパターン:** 制約のない単純なパターンは、指定したタイプのファクトに一致します。たとえば、次は、申込者が存在することだけが条件になります。

```
when
  Applicant( )
```

   - **制約のあるパターン:** 制約を持つパターンは、指定したタイプのファクトと、追加制限を括弧で指定したパターン (true または false) に一致します。たとえば、次は、申込者が 21 歳に満たないことを条件としています。

```
when
  Applicant( age < 21 )
```

- **バインディングのあるパターン:** パターンのバインディングは簡単な参照となり、ルールのその他のコンポーネントが、定義したバターンに戻って参照します。たとえば、次の例では、**LoanApplication** のバインディング **a** が、underage の申込者に関連するアクションとして使用されます。

```
when
  a : LoanApplication( )
  Applicant( age < 21 )
then
  a.setApproved( false );
  a.setExplanation( "Underage" )
```

3. 引き続き、このルールに適用する条件パターンをすべて定義します。以下は、DRL 条件を定義するいくつかのキーワードオプションです。

- **and**: 条件コンポーネントを論理積に分類します。接中辞および接頭辞の **and** がサポートされます。デフォルトでは、結合演算子を指定しないと、リストされている条件またはアクションがすべて **and** と結合します。

```
a : LoanApplication( ) and Applicant( age < 21 )

a : LoanApplication( )
and Applicant( age < 21 )

a : LoanApplication( )
Applicant( age < 21 )

// All of the above are the same.
```

- **or**: 条件コンポーネントを論理和に分類します。接中辞および接頭辞の **or** がサポートされます。

```
Bankruptcy( amountOwed == 100000 ) or IncomeSource( amount ==
20000 )

Bankruptcy( amountOwed == 100000 )
or IncomeSource( amount == 20000 )
```

- **exists**: 存在すべきファクトおよび制約を指定します。これは、ファクトが存在していることを示しているのではなく、ファクトが存在すべきであることを示しています。このオプションは、最初に一致したものだけが適用され、その後一致するものは無視されます。

```
exists ( Bankruptcy( yearOfOccurrence > 1990 || amountOwed >
10000 ) )
```

- **not**: 存在するべきでないファクトおよび制約を指定します。

```
not ( Applicant( age < 21 ) )
```

- **forall**: 最初のパターンに一致したすべてのファクトが残りのパターンに一致する制約を作成します。

```
forall( app : Applicant( age < 21 )
              Applicant( this == app, status = 'underage' ) )
```

- **from**: 条件パターンによりデータが一致するソースを指定します。

```
Applicant( ApplicantAddress : address )
Address( zipcode == "23920W" ) from ApplicantAddress
```

- **entry-point**: パターンのデータソースに対応するエントリーポイントを定義します。通常は **from** とともに使用します。

```
Applicant( ) from entry-point "LoanApplication"
```

- **collect**: 構成を条件の一部として使用できる、オブジェクトのコレクションを定義します。この例では、指定した各担保に対して、プロセスエンジンで保留されているすべての申し込みが **ArrayLists** に分類されます。申し込みが 3 つ以上ある場合は、このルールが実行します。

```
m : Mortgage()
a : ArrayList( size >= 3 )
    from collect( LoanApplication( Mortgage == m, status ==
'pending' ) )
```

- **accumulate**: オブジェクトのコレクションを処理し、各要素のカスタムアクションを実行し、(制約が **true** と評価されると) 結果オブジェクトを 1 つ以上返します。このオプションは、**collect** よりも強力で、柔軟性が高いオプションです。**accumulate(<source pattern>; <functions> [;<constraints>])** 形式を使用します。この例では、**min**、**max**、および **average** は累積関数で、各センサーのすべての測定値から、最低気温、最高気温、そして平均気温の値を計算します。その他のサポートされる関数には、**count**、**sum**、**variance**、**standardDeviation**、**collectList**、および **collectSet** があります。

```
s : Sensor()
accumulate( Reading( sensor == s, temp : temperature );
            min : min( temp ),
            max : max( temp ),
            avg : average( temp );
            min < 20, avg > 70 )
```

高度な DRL オプション

これは、条件を定義する基本的なキーワードオプションおよびパターン構築の例です。さらに高度な DRL オプションと構文が DRL デザイナーでサポートされています。オンラインの『Drools ドキュメンテーション』を参照してください。

4. ルールの条件コンポーネントをすべて定義したら、DRL デザイナーの右上のツールバーの **Validate** をクリックして、DRL ファイルの妥当性を確認します。ファイルの妥当性確認に失敗したら、エラーメッセージに記載された問題に対応し、DRL ファイルの構文およびコンポーネントをすべて見直し、エラーが表示されなくなるまで再度、ファイルを検証します。

5. DRL デザイナーで **Save** をクリックして、設定した内容を保存します。

4.2. DRL ルールへの THEN アクションの追加

ルールの **then** 部分には、ルールの条件部分に一致したときに実行するアクションが含まれます。たとえば、ローンの申込者が 21 歳に満たない場合は、**Underage** ルールの **then** アクションが **setApproved(false)** となり、年齢が基準に達していないためローンの申し込みが承認されません。アクションは、ルールの条件と、パッケージで利用可能オブジェクトに基づいて結果を実行します。

前提条件

- **package** は DRL ファイルに定義されます。これは、ファイルの作成時に行われます。
- ルールで使用したデータオブジェクトの **import** リストが、DRL ファイルの **package** 行の下に定義されます。データオブジェクトは、このパッケージ、または別の Business Central パッケージから使用できます。
- **rule** 名は、**package**、**import**、または DRL ファイル全体に適用されるその他の行の下に、**rule "name"** という形式で定義されます。同じパッケージでルール名を重複させることはできません。ルールの動作 (**salience**、**no-loop** など) を定義する任意のルール属性は、ルール名の下、**when** セクションの前に定義します。

手順

1. DRL デザイナーで、ルールの **when** セクションの後に **then** を入力して、アクション命令文を追加します。
2. ルールの条件に基づいて、ファクトパターンに対して実行するアクションを 1 つ以上入力します。
   次は、DRL アクションを定義するキーワードオプションの例です。
   - **and**: アクションコンポーネントを論理積に分類します。接中辞および接頭辞の **and** がサポートされます。デフォルトでは、結合演算子を指定しないと、リストされている条件またはアクションがすべて **and** と結合します。

     ```
     application.setApproved ( false ) and application.setExplanation(
     "has been bankrupt" );

     application.setApproved ( false );
     and application.setExplanation( "has been bankrupt" );

     application.setApproved ( false );
     application.setExplanation( "has been bankrupt" );

     // All of the above are the same.
     ```

   - **set**: フィールドの値を設定します。

     ```
     application.setApproved ( false );
     application.setExplanation( "has been bankrupt" );
     ```

   - **modify**: ファクトに対して修正するフィールドを指定し、変更をプロセスエンジンに通知します。

     ```
     modify( LoanApplication ) {
             setAmount( 100 )
     }
     ```

- **update**: フィールドと、修正される関連ファクト全体を指定して、その変更をプロセスエンジンに通知します。ファクトが変更したら、アップデートした値の影響を受ける可能性がある別のファクトを変更する前に、**update** を呼び出す必要があります。**modify** キーワードには、この追加手順がありません。

```
update( LoanApplication ) {
        setAmount( 100 )
}
```

- **delete**: プロセスエンジンからオブジェクトを削除します。キーワード **retract** も DRL デザイナーでサポートされ、同じアクションを実行しますが、キーワード **insert** との一貫性を保つために **delete** が推奨されます。

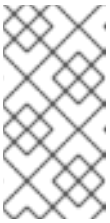
```
delete( LoanApplication );
```

- **insert**: **新しい** ファクトを挿入し、ファクトに必要な結果フィールドと値を定義します。

```
insert( new Applicant() );
```

- **insertLogical**: **新しい** ファクトをプロセスエンジンに論理的に挿入し、ファクトに必要な結果フィールドと値を追加します。Red Hat Process Automation Manager のプロセスエンジンは、ファクトの挿入および取り消しに対して論理的な決断を行います。定期的な挿入、または指定した挿入の後に、ファクトを明示的に取り消す必要があります。論理挿入の後に、ファクトをアサートした条件が TRUE ではなくなると、ファクトは自動的に取り消されます。

```
insertLogical( new Applicant() );
```



高度な DRL オプション

これは、アクションを定義する基本的なキーワードオプションおよびパターン構築の例です。さらに高度な DRL オプションと構文が DRL デザイナーでサポートされています。オンラインの『Drools ドキュメンテーション』を参照してください。

3. ルールのアクションコンポーネントをすべて定義したら、DRL デザイナーの右上のツールバーの **Validate** をクリックして、DRL ファイルの妥当性を確認します。ファイルの妥当性確認に失敗したら、エラーメッセージに記載された問題に対応し、DRL ファイルの構文およびコンポーネントをすべて見直し、エラーが表示されなくなるまで再度、ファイルを検証します。

4. DRL デザイナーで **Save** をクリックして、設定した内容を保存します。

4.2.1. ルールの属性

ルール属性は、ルールの動作を修正するビジネスルールを指定する追加設定です。次の表では、ルールに割り当て可能な属性の名前と、対応する値を紹介します。

表4.1 ルールの属性

属性	値

属性	値
salience	ルールの優先順位を定義する整数。ルールの salience 値を高くすると、アクティベーションキューに追加したときの優先順位が高くなります。 例: **salience 10**
enabled	ブール値。このオプションを選択すると、ルールが有効になります。このオプションを選択しないと、ルールは無効になります。 例: **enabled true**
date-effective	日付定義および時間定義を含む文字列。現在の日時が **date-effective** 属性よりも後の場合は、このルールがアクティブになります。 例: **date-effective "4-Sep-2018"**
date-expires	日時定義を含む文字列。現在日時が **date-expires** 属性よりも後になると、このルールをアクティブにすることはできません。 例: **date-expires "4-Oct-2018"**
no-loop	ブール値。このオプションを設定すると、以前一致した条件がこのルールにより再トリガーとなる場合に、このルールを再度アクティブにする (ループする) ことができません。条件を選択しないと、この状況でルールがループされます。 例: **no-loop true**
agenda-group	ルールを割り当てるアジェンダグループを指定する文字列。アジェンダグループを使用すると、アジェンダをパーティションで区切り、ルールのグループに対する実行をさらに制御できます。フォーカスを取得したアジェンダグループのルールだけがアクティブになります。 例: **agenda-group "GroupName"**
activation-group	ルールを割り当てるアクティベーション (または XOR) グループを指定する文字列。アクティベーショングループでアクティブにできるルールは 1 つだけです。最初のルールが実行されると、アクティベーショングループの中で、アクティベーションが保留中のルールはすべてキャンセルされます。 例: **activation-group "GroupName"**
duration	ルールの条件が一致している場合に、ルールがアクティブになってからの時間をミリ秒で定義する長整数値。 例: **duration 10000**
timer	ルールのスケジュールに対する **int** (間隔) または **cron** タイマー定義を指定する文字列。 例: **timer "*/5 * * * *"** (5 分ごと)

属性	値
calendar	ルールのスケジュールを指定する Quartz カレンダーの定義。 例: **calendars "* * 0-7,18-23 ? * *"** (営業時間外を除く)
auto-focus	アジェンダグループ内のルールにのみ適用可能なブール値。このオプションが選択されている場合は、次にルールがアクティブになると、そのルールが割り当てられたアジェンダグループに自動的にフォーカスが移ります。 例: **auto-focus true**
lock-on-active	ルールフローグループまたはアジェンダグループ内のルールにのみ適用可能なブール値。このオプションを選択すると、次回、ルールのルールフローグループがアクティブになるか、ルールのアジェンダグループがフォーカスを受け取ると、(ルールフローグループがアクティブでなくなるか、アジェンダグループがフォーカスを失うまで) ルールをアクティブにすることができません。これは、**no-loop** 属性を強力にしたものです。なぜなら、一致するルールのアクティベーションが、(ルールそのものによるものだけでなく) アップデート元にかかわらず破棄されるためです。この属性は、ファクトを修正するルールが多数あり、ルールの再一致と再発行を希望しない計算ルールに適しています。 例: **lock-on-active true**
ruleflow-group	ルールフローグループを指定する文字列。ルールフローグループで、関連するルールフローによってそのグループがアクティブになった場合に限りルールを発行できます。 例: **ruleflow-group "GroupName"**
dialect	ルールのコード表記に使用される言語を指定する文字列 (**JAVA** または **MVEL**)。デフォルトでは、ルールは、パッケージレベルに指定されている方言を使用します。ここで指定した方言は、ルールのパッケージ方言設定を上書きします。 例: **dialect "JAVA"**

第5章 ルールの実行

ルールの例を特定するか、Business Central でルールを作成したら、関連のプロジェクトをビルドしてデプロイし、ローカルまたは Process Server でルールを実行してルールをテストできます。

前提条件

- Business Central および Process Server がインストールされて実行されている。インストールオプションは**『Red Hat Process Automation Manager インストールの計画』**を参照してください。

手順

1. Business Central で、**Menu** → **Design** → **Projects** に移動して、プロジェクト名をクリックします。

2. プロジェクトの **Assets** ページの右上にある **Deploy** をクリックして、プロジェクトをビルドして Process Server にデプロイします。ビルドに失敗したら、画面下部の **Alerts** パネルに記載されている問題に対処します。
プロジェクトのデプロイに関する詳細は**『Packaging and deploying a Red Hat Process Automation Manager project』**を参照してください。

3. ローカルでのルール実行に使用するか、Business Server でルールを実行するクライアントアプリケーションとして使用できるように、まだ作成されていない場合には、Process Central 外に Maven または Java プロジェクトを作成します。プロジェクトには、**`pom.xml`** ファイルと、プロジェクトリソースの実行に必要なその他のコンポーネントを含める必要があります。
テストプロジェクトの例については、「Other methods for creating and executing DRL rules」を参照してください。

4. テストプロジェクトまたはクライアントアプリケーションの **`pom.xml`** ファイルを開き、以下の依存関係が追加されていない場合は追加します。

   - **`kie-ci`**: クライアントアプリケーションで、**`ReleaseId`** を使用して、Business Central プロジェクトデータをローカルにロードします。
   - **`kie-server-client`**: クライアントアプリケーションで、Process Server のアセットを使用してリモートに接続します。
   - **`slf4j`**: (オプション) クライアントアプリケーションで、Process Server に接続したあと、SLF4J (Simple Logging Facade for Java) を使用して、デバッグのログ情報を返します。

   クライアントアプリケーションの **`pom.xml`** ファイルにおける、Red Hat Process Automation Manager 7.2 の依存関係の例

```
// For local execution:
<dependency>
  <groupId>org.kie</groupId>
  <artifactId>kie-ci</artifactId>
  <version>7.14.0.Final-redhat-00004</version>
</dependency>

// For remote execution on Process Server:
<dependency>
  <groupId>org.kie.server</groupId>
  <artifactId>kie-server-client</artifactId>
```

```
  <version>7.14.0.Final-redhat-00004</version>
</dependency>

// For debug logging (optional):
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-simple</artifactId>
  <version>1.7.25</version>
</dependency>
```

このアーティファクトで利用可能なバージョンについては、オンラインの Nexus Repository Manager でグループ ID とアーティファクト ID を検索してください。

注記

個別の依存関係に対して Red Hat Process Automation Manager **<version>** を指定するのではなく、Red Hat Business Automation 部品表 (BOM) の依存関係をプロジェクトの **pom.xml** ファイルに追加することを検討してください。Red Hat Business Automation BOM は、Red Hat Decision Manager と Red Hat Process Automation Manager の両方に適用します。BOM ファイルを追加すると、指定の Maven リポジトリーからの一時的な依存関係の内、正しいバージョンが、このプロジェクトに追加されます。

BOM 依存関係の例:

```
<dependency>
  <groupId>com.redhat.ba</groupId>
  <artifactId>ba-platform-bom</artifactId>
  <version>7.2.0.GA-redhat-00004</version>
  <scope>import</scope>
  <type>pom</type>
</dependency>
```

Red Hat Business Automation BOM (Bill of Materials) についての詳細情報は、「What is the mapping between Red Hat Process Automation Manager and the Maven library version?」を参照してください。

5. モジュールクラスを含むアーティファクトの依存関係が、クライアントアプリケーションの **pom.xml** ファイルに定義されていて、デプロイしたプロジェクトの **pom.xml** ファイルに記載されているのと同じであることを確認します。モデルクラスの依存関係が、クライアントアプリケーションとプロジェクトで異なると、実行エラーが発生します。
Business Central でプロジェクトの **pom.xml** を利用するには、プロジェクトで既存のアセットを選択し、画面左側の **Project Explorer** メニューで **Customize View** ギアアイコンをクリックし、**Repository View** → **pom.xml** を選択します。

   たとえば、以下の **Person** クラスの依存関係は、クライアントと、デプロイしたプロジェクトの **pom.xml** ファイル両方に表示されます。

```
<dependency>
  <groupId>com.sample</groupId>
  <artifactId>Person</artifactId>
  <version>1.0.0</version>
</dependency>
```

6. デバッグ向けロギングを行うために、**slf4j** 依存関係を、クライアントアプリケーションの **pom.xml** ファイルに追加した場合は、関連するクラスパス (Maven の **src/main/resources/META-INF** 内など) に **simplelogger.properties** ファイルを作成し、以下の内容を記載します。

```
org.slf4j.simpleLogger.defaultLogLevel=debug
```

7. クライアントアプリケーションに、必要なインポートを含む **.java** メインクラスと、KIE ベースをロードする **main()** メソッドを作成し、ファクトを挿入し、ルールを実行します。
たとえば、プロジェクトの **Person** オブジェクトには、名前、苗字、時給、賃金を設定および取得するゲッターメソッドおよびセッターメソッドが含まれます。プロジェクトにある以下の **Wage** ルールでは、賃金と時給を計算し、その結果に基づいてメッセージを表示します。

```
package com.sample;

import com.sample.Person;

dialect "java"

rule "Wage"
  when
    Person(hourlyRate * wage > 100)
    Person(name : firstName, surname : lastName)
  then
    System.out.println("Hello" + " " + name + " " + surname + "!");
    System.out.println("You are rich!");
end
```

(必要に応じて) Process Server の外でローカルにこのルールをテストするには、**.java** クラスで、KIE サービス、KIE コンテナー、および KIE セッションをインポートするように設定し、その後、**main()** メソッドを使用して、定義したファクトモデルに対してすべてのルールを実行するようにします。

ローカルでルールの実行

```
import org.kie.api.KieServices;
import org.kie.api.runtime.KieContainer;
import org.kie.api.runtime.KieSession;

public class RulesTest {

  public static final void main(String[] args) {
    try {
      // Identify the project in the local repository:
      ReleaseId rid = new ReleaseId();
      rid.setGroupId("com.myspace");
      rid.setArtifactId("MyProject");
      rid.setVersion("1.0.0");

      // Load the KIE base:
      KieServices ks = KieServices.Factory.get();
      KieContainer kContainer = ks.newKieContainer(rid);
      KieSession kSession = kContainer.newKieSession();
```

```
        // Set up the fact model:
        Person p = new Person();
        p.setWage(12);
        p.setFirstName("Tom");
        p.setLastName("Summers");
        p.setHourlyRate(10);

        // Insert the person into the session:
        kSession.insert(p);

        // Fire all rules:
        kSession.fireAllRules();
        kSession.dispose();
      }

      catch (Throwable t) {
        t.printStackTrace();
      }
    }
}
```

Process Server でこのルールをテストするには、ローカル例と同じように、インポートとルール実行情報で **.java** クラスを設定し、KIE サービス設定および KIE サービスクライアントの詳細を指定します。

Process Server でルールの実行

```
package com.sample;

import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set;

import org.kie.api.command.BatchExecutionCommand;
import org.kie.api.command.Command;
import org.kie.api.KieServices;
import org.kie.api.runtime.ExecutionResults;
import org.kie.api.runtime.KieContainer;
import org.kie.api.runtime.KieSession;
import org.kie.server.api.marshalling.MarshallingFormat;
import org.kie.server.api.model.ServiceResponse;
import org.kie.server.client.KieServicesClient;
import org.kie.server.client.KieServicesConfiguration;
import org.kie.server.client.KieServicesFactory;
import org.kie.server.client.RuleServicesClient;

import com.sample.Person;

public class RulesTest {

  private static final String containerName = "testProject";
  private static final String sessionName = "myStatelessSession";

  public static final void main(String[] args) {
    try {
```

```
      // Define KIE services configuration and client:
      Set<Class<?>> allClasses = new HashSet<Class<?>>();
      allClasses.add(Person.class);
      String serverUrl = "http://$HOST:$PORT/kie-
server/services/rest/server";
      String username = "$USERNAME";
      String password = "$PASSWORD";
      KieServicesConfiguration config =
        KieServicesFactory.newRestConfiguration(serverUrl,
                                                username,
                                                password);
      config.setMarshallingFormat(MarshallingFormat.JAXB);
      config.addExtraClasses(allClasses);
      KieServicesClient kieServicesClient =
        KieServicesFactory.newKieServicesClient(config);

      // Set up the fact model:
      Person p = new Person();
      p.setWage(12);
      p.setFirstName("Tom");
      p.setLastName("Summers");
      p.setHourlyRate(10);

      // Insert Person into the session:
      KieCommands kieCommands =
KieServices.Factory.get().getCommands();
      List<Command> commandList = new ArrayList<Command>();
      commandList.add(kieCommands.newInsert(p, "personReturnId"));

      // Fire all rules:

commandList.add(kieCommands.newFireAllRules("numberOfFiredRules"));
      BatchExecutionCommand batch =
kieCommands.newBatchExecution(commandList, sessionName);

      // Use rule services client to send request:
      RuleServicesClient ruleClient =
kieServicesClient.getServicesClient(RuleServicesClient.class);
      ServiceResponse<ExecutionResults> executeResponse =
ruleClient.executeCommandsWithResults(containerName, batch);
      System.out.println("number of fired rules:" +
executeResponse.getResult().getValue("numberOfFiredRules"));
    }

    catch (Throwable t) {
      t.printStackTrace();
    }
  }
}
```

8. 設定した **.java** クラスをプロジェクトディレクトリーから実行します。(Red Hat JBoss Developer Studio などの) 開発プラットフォーム、またはコマンドラインでファイルを実行できます。
(プロジェクトディレクトリーにおける) Maven の実行例:

```
mvn clean install exec:java -
```

```
Dexec.mainClass="com.sample.app.RulesTest"
```

(プロジェクトディレクトリーにおける) Java の実行例

```
javac -classpath "./$DEPENDENCIES/*:." RulesTest.java
java -classpath "./$DEPENDENCIES/*:." RulesTest
```

9. コマンドラインおよびサーバーログで、ルール実行のステータスを確認します。ルールが期待通りに実行しない場合は、プロジェクトに設定したルールと、メインのクラス設定を確認して、提供されるデータの妥当性を確認します。

第6章 その他の DRL ルールの作成および実行方法

Business Central インターフェースに DRL ルールを作成して管理する代わりに、Red Hat Developer Studio、Java オブジェクト、または Maven アーキタイプを使用して、外部のスタンドアロンプロジェクトに DRL ルールファイルを作成できます。このスタンドアロンプロジェクトは、ナレッジ JAR (kJAR) 依存関係として、Business Central の既存の Red Hat Process Automation Manager プロジェクトに統合できます。スタンドアロンプロジェクトの DRL ファイルには、少なくても、必要な **`package`** 仕様、**`import`** リスト、および **`rule`** 定義が含まれる必要があります。グローバル変数や関数など、その他の DRL コンポーネントは任意です。DRL ルールに関連するすべてのデータオブジェクトは、スタンドアロンの DRL プロジェクトまたはデプロイメントに含まれる必要があります。

Maven または Java プロジェクトで実行可能なルールモデルを使用して、ビルド時に実行するルールセットの Java ベース表記を提供します。実行可能モデルは Red Hat Process Automation Manager の標準アセットパッケージングの代わりとなるもので、より効率的です。KIE コンテナーと KIE ベースの作成がより迅速にでき、DRL (Drools Rule Language) ファイルリストや他の Red Hat Process Automation Manager アセットが多い場合は、特に有効です。

6.1. RED HAT JBOSS DEVELOPER STUDIO への DRL ルールの作成および実行

Red Hat JBoss Developer Studio を使用して、ルールを持つ DRL ファイルを作成し、Red Hat Process Automation Manager デシジョンサービスにファイルを統合します。DRL ルールを作成する方法は、デシジョンサービスに Red Hat Developer Studio を使用する場合や、同じワークフローを継続したい場合に便利です。この方法を使用していない場合は、Red Hat Process Automation Manager の代わりに Business Central インターフェースを使用して、DRL ファイルや、その他のルールアセットを作成することが推奨されます。

前提条件/事前作業

Red Hat カスタマーポータル から Red Hat JBoss Developer Studio をインストールしています。

手順

1. Red Hat JBoss Developer Studio で、**File** → **New** → **Project** をクリックします。
2. 開いた **New Project** ウィンドウで、**Drools** → **Drools Project** を選択し、**Next** をクリックします。
3. **Create a project and populate it with some example files to help you get started quickly** の 2 番目のアイコンをクリックして、**Next** をクリックします。
4. **Project name** を入力し、プロジェクトのビルドオプションで **Maven** ラジオボタンを選択します。GAV 値が自動的に生成されます。必要に応じて、プロジェクトに対してこの値を更新できます。
   - **Group ID:** **`com.sample`**
   - **Artifact ID:** **`my-project`**
   - **Version:** **`1.0.0-SNAPSHOT`**
5. **Finish** をクリックしてプロジェクトを作成します。
   これは、基本的なプロジェクト構造、クラスパス、サンプルルールを設定します。以下は、プロジェクト構造の概要です。

```
my-project
 `-- src/main/java
    | `-- com.sample
    |    `-- DecisionTableTest.java
    |    `-- DroolsTest.java
    |    `-- ProcessTest.java
    |
 `-- src/main/resources
    | `-- dtables
    |    `-- Sample.xls
    | `-- process
    |    `-- sample.bpmn
    | `-- rules
    |    `-- Sample.drl
    | `-- META-INF
    |
 `-- JRE System Library
    |
 `-- Maven Dependencies
    |
 `-- Drools Library
    |
 `-- src
    |
 `-- target
    |
 `-- pom.xml
```

以下の要素に注目してください。

- **src/main/resources** ディレクトリーの **Sample.drl** ルールファイル。これには、サンプルの **Hello World** ルールおよび **GoodBye** ルールが含まれます。
- **com.sample** パッケージの **src/main/java** ディレクトリーにある **DroolsTest.java** ファイル。**Sample.drl** ルールの実行には、**DroolsTest** クラスを使用できます。
- 実行するのに必要な JAR ファイルを含むカスタムのクラスパスとなる **Drools Library** ディレクトリー。

既存の **Sample.drl** ファイルおよび **DroolsTest.java** ファイルを必要に応じて新しい設定に変更するか、ルールファイルをオブジェクトファイルを新たに作成します。この手順では、ルールと Java オブジェクトを新たに作成します。

6. ルールが有効な Java オブジェクトを作成します。
この例では、 **my-project/src/main/java/com.sample** に **Person.java** ファイルが作成されます。**Person** クラスには、名前、苗字、時給、賃金を設定および取得するゲッターメソッドおよびセッターメソッドが含まれます。

```
 public class Person {
   private String firstName;
   private String lastName;
   private Integer hourlyRate;
   private Integer wage;

   public String getFirstName() {
```

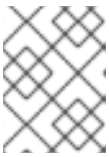
```
      return firstName;
    }

    public void setFirstName(String firstName) {
      this.firstName = firstName;
    }

    public String getLastName() {
      return lastName;
    }

    public void setLastName(String lastName) {
      this.lastName = lastName;
    }

    public Integer getHourlyRate() {
      return hourlyRate;
    }

    public void setHourlyRate(Integer hourlyRate) {
      this.hourlyRate = hourlyRate;
    }

    public Integer getWage(){
      return wage;
    }

    public void setWage(Integer wage){
      this.wage = wage;
    }
  }
```

7. **File** → **Save** をクリックして、ファイルを保存します。

8. **my-project/src/main/resources/rules** に、**.drl** 形式のルールファイルを作成します。DRL ファイルには、少なくともパッケージの指定と、(1 つまたは複数の) ルールで使用されるデータオブジェクトのインポートリストと、**when** 条件および **then** アクションを持つ 1 つ以上のルールが含まれます。
   以下の **Wage.drl** ファイルには、**Person** クラスをインポートする **Wage** ルールが含まれ、賃金および時給の値を計算し、その結果に基づいてメッセージを表示します。

```
package com.sample;

import com.sample.Person;

dialect "java"

rule "Wage"
  when
    Person(hourlyRate * wage > 100)
    Person(name : firstName, surname : lastName)
  then
    System.out.println("Hello" + " " + name + " " + surname + "!");
    System.out.println("You are rich!");
end
```

9. **File** → **Save** をクリックして、ファイルを保存します。
10. メインクラスを作成し、Java オブジェクトを作成したディレクトリーに保存します。メインクラスは KIE ベースをロードし、ルールを実行します。



注記

また、**DroolsTest.java** サンプルファイルと同様に、**main()** メソッドと **Person** クラスを 1 つの Java オブジェクトファイルに追加できます。

11. メインクラスに、KIE サービス、KIE コンテナー、および KIE セッションをインポートするのに必要な **import** 命令文を追加します。つぎに、KIE ベースをロードし、ファクトを挿入し、ファクトモデルをルールに渡す **main()** メソッドからルールを実行します。
この例では、必要なインポートと **main()** メソッドを使用して、**my-project/src/main/java/com.sample** に **RulesTest.java** ファイルを作成します。

```
package com.sample;

import org.kie.api.KieServices;
import org.kie.api.runtime.KieContainer;
import org.kie.api.runtime.KieSession;

public class RulesTest {
  public static final void main(String[] args) {
    try {
      // Load the KIE base:
      KieServices ks = KieServices.Factory.get();
      KieContainer kContainer = ks.getKieClasspathContainer();
      KieSession kSession = kContainer.newKieSession();

      // Set up the fact model:
      Person p = new Person();
      p.setWage(12);
      p.setFirstName("Tom");
      p.setLastName("Summers");
      p.setHourlyRate(10);

      // Insert the person into the session:
      kSession.insert(p);

      // Fire all rules:
      kSession.fireAllRules();
      kSession.dispose();
    }

    catch (Throwable t) {
      t.printStackTrace();
    }
  }
}
```

12. **File** → **Save** をクリックして、ファイルを保存します。
13. プロジェクトで DRL アセットをすべて作成して保存したあと、プロジェクトフォルダーを右ク

リックして、**Run As** → **Java Application** を選択してプロジェクトをビルドします。プロジェクトのビルドに失敗したら、Developer Studio の下部ウィンドウの **Problems** タブに記載されている問題に対応し、プロジェクトビルドされるまで妥当性確認を行います。

RUN AS → JAVA APPLICATION オプションが利用できない場合

プロジェクトを右クリックして、**Run As** を選択した場合に **Java Application** が選択肢にない場合は、**Run As** → **Run Configurations** に移動して **Java Application** を右クリックし、**New** をクリックします。次に、**Main** タブで、**Project** と、関連する **Main class** を参照して選択します。**Apply** をクリックし、**Run** をクリックしてプロジェクトをテストします。再度プロジェクトフォルダーを右クリックすると、**Java Application** オプションが表示されます。

Red Hat Process Automation Manager で既存のプロジェクトと新しいルールアセットを統合するには、ナレッジJAR (KJAR) として新規プロジェクトをコンパイルし、Business Central でプロジェクトの **pom.xml** ファイルに、依存関係としてこのプロジェクトを追加します。Business Central でプロジェクト **pom.xml** にアクセスするには、プロジェクトで既存のアセットを選択し、画面左側の **Project Explorer** メニューで **Customize View** ギアアイコンをクリックし、**Repository View** → **pom.xml** を選択します。

6.2. JAVA を使用した DRL ルールの作成および実行

Java オブジェクトを使用して、ルールを持つ DRL ファイルを作成し、オブジェクトを Red Hat Process Automation Manager デシジョンサービスに統合します。DRL ルールを作成する方法は、デシジョンサービスに外部 Java オブジェクトを使用している場合や、同じワークフローを継続したい場合に便利です。この方法を使用しなくなった場合は、Red Hat Process Automation Manager の Business Central インターフェースを使用して、DRL ファイルや、その他のルールアセットを作成することが推奨されます。

手順

1. ルールが有効な Java オブジェクトを作成します。
この例では、**my-project** ディレクトリーに **Person.java** ファイルが作成されます。**Person** クラスには、名前、苗字、時給、賃金を設定および取得するゲッターメソッドおよびセッターメソッドが含まれます。

```
public class Person {
  private String firstName;
  private String lastName;
  private Integer hourlyRate;
  private Integer wage;

  public String getFirstName() {
    return firstName;
  }

  public void setFirstName(String firstName) {
    this.firstName = firstName;
  }

  public String getLastName() {
    return lastName;
  }
```

```
    public void setLastName(String lastName) {
      this.lastName = lastName;
    }

    public Integer getHourlyRate() {
      return hourlyRate;
    }

    public void setHourlyRate(Integer hourlyRate) {
      this.hourlyRate = hourlyRate;
    }

    public Integer getWage(){
      return wage;
    }

    public void setWage(Integer wage){
      this.wage = wage;
    }
  }
```

2. **my-project** ディレクトリーに、**.drl** 形式のルールファイルを作成します。DRL ファイルには、少なくともパッケージの指定と、(1 つまたは複数の) ルールで使用されるデータオブジェクトのインポートリストと、**when** 条件および **then** アクションを持つ 1 つ以上のルールが含まれます。
   以下の **Wage.drl** ファイルには、賃金と時給を計算し、その結果に基づいてメッセージを表示する **Wage** ルールが含まれています。

```
package com.sample;

import com.sample.Person;

dialect "java"

rule "Wage"
  when
    Person(hourlyRate * wage > 100)
    Person(name : firstName, surname : lastName)
  then
    System.out.println("Hello" + " " + name + " " + surname + "!");
    System.out.println("You are rich!");
end
```

3. メインクラスを作成し、Java オブジェクトを作成したディレクトリーに保存します。メインクラスは KIE ベースをロードし、ルールを実行します。

4. メインクラスに、KIE サービス、KIE コンテナー、および KIE セッションをインポートするのに必要な **import** 命令文を追加します。つぎに、KIE ベースをロードし、ファクトを挿入し、ファクトモデルをルールに渡す **main()** メソッドからルールを実行します。
   この例では、必要なインポートと **main()** メソッドを使用して、**my-project** に **RulesTest.java** ファイルを作成します。

```
import org.kie.api.KieServices;
import org.kie.api.runtime.KieContainer;
import org.kie.api.runtime.KieSession;
```

```
public class RulesTest {
  public static final void main(String[] args) {
    try {
      // Load the KIE base:
      KieServices ks = KieServices.Factory.get();
      KieContainer kContainer = ks.getKieClasspathContainer();
      KieSession kSession = kContainer.newKieSession();

      // Set up the fact model:
      Person p = new Person();
      p.setWage(12);
      p.setFirstName("Tom");
      p.setLastName("Summers");
      p.setHourlyRate(10);

      // Insert the person into the session:
      kSession.insert(p);

      // Fire all rules:
      kSession.fireAllRules();
      kSession.dispose();
    }

    catch (Throwable t) {
      t.printStackTrace();
    }
  }
}
```

5. Red Hat カスタマーポータル から **Red Hat Process Automation Manager 7.2.0 Source Distribution** の ZIP ファイルをダウンロードし、**my-project/pam-engine-jars/** で展開します。

6. **my-project/META-INF** ディレクトリーに、以下の内容の **kmodule.xml** メタデータファイルを作成します。

```
<?xml version="1.0" encoding="UTF-8"?>
<kmodule xmlns="http://www.drools.org/xsd/kmodule">
</kmodule>
```

この **kmodule.xml** ファイルは、KIE ベースへのリソースを選択し、セッションを設定する KIE モジュールの記述子です。このファイルを使用すると、KIE ベースを 1 つ以上定義して設定し、特定の KIE ベースの特定の **packages** から DRL ファイルを含めることができます。各 KIE ベースから KIE セッションを 1 つ以上作成することもできます。

次の例は、より高度な **kmodule.xml** ファイルを表示します。

```
<?xml version="1.0" encoding="UTF-8"?>
<kmodule xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://www.drools.org/xsd/kmodule">
  <kbase name="KBase1" default="true" eventProcessingMode="cloud"
equalsBehavior="equality" declarativeAgenda="enabled"
packages="org.domain.pkg1">
    <ksession name="KSession1_1" type="stateful" default="true" />
```

```
    <ksession name="KSession1_2" type="stateful" default="true"
beliefSystem="jtms" />
  </kbase>
  <kbase name="KBase2" default="false" eventProcessingMode="stream"
equalsBehavior="equality" declarativeAgenda="enabled"
packages="org.domain.pkg2, org.domain.pkg3" includes="KBase1">
    <ksession name="KSession2_1" type="stateless" default="true"
clockType="realtime">
      <fileLogger file="debugInfo" threaded="true" interval="10" />
      <workItemHandlers>
        <workItemHandler name="name" type="new
org.domain.WorkItemHandler()" />
      </workItemHandlers>
      <listeners>
        <ruleRuntimeEventListener
type="org.domain.RuleRuntimeListener" />
        <agendaEventListener type="org.domain.FirstAgendaListener"
/>
        <agendaEventListener type="org.domain.SecondAgendaListener"
/>
        <processEventListener type="org.domain.ProcessListener" />
      </listeners>
    </ksession>
  </kbase>
</kmodule>
```

この例は、KIE ベースを 2 つ定義します。KIE ベース **KBase1** から 2 つの KIE セッションをインスタンス化し、**KBase2** セッションから 1 つの KIE セッションをインスタンス化します。**KBase2** の KIE セッションは、**ステートレス**な KIE セッションですが、これは 1 つ前の KIE セッションで呼び出されたデータ (1 つ前のセッションの状態) が、セッションの呼び出しと呼び出しの間で破棄されることを示しています。ルールアセットで特定の **packages** が両方の KIE ベースに含まれます。この方法でパッケージを指定した場合は、指定したパッケージを反映するフォルダー構造で DRL ファイルを整理する必要があります。

7. Java オブジェクトですべての DRL アセットを作成して保存したあと、コマンドラインで **my-project** ディレクトリーに移動し、以下のコマンドを実行して Java ファイルをビルドします。**RulesTest.java** を、Java のメインクラスの名前に置き換えます。

```
javac -classpath "./pam-engine-jars/*:." RulesTest.java
```

ビルドに失敗したら、コマンドラインのエラーメッセージに記載されている問題に対応し、エラーが表示されなくなるまで Java オブジェクトの妥当性確認を行います。

8. Java ファイルが問題なくビルトできたら、以下のコマンドを実行してローカルでルールを実行します。**RulesTest** を、Java のメインクラスの接頭辞に置き換えます。

```
java -classpath "./pam-engine-jars/*:." RulesTest
```

9. ルールを見直して、適切に実行したことを確認し、Java ファイルで必要な変更に対応します。

Red Hat Process Automation Manager で既存のプロジェクトと新しいルールアセットを統合するには、ナレッジJAR (KJAR) として新規 Java プロジェクトをコンパイルし、Business Central でプロジェクトの **pom.xml** ファイルに、依存関係としてこのプロジェクトを追加します。Business Central でプ

ロジェクト **pom.xml** にアクセスするには、プロジェクトで既存のアセットを選択し、画面左側の **Project Explorer** メニューで **Customize View** ギアアイコンをクリックし、**Repository View** → **pom.xml** を選択します。

6.3. MAVEN を使用した DRL ルールの作成および実行

Maven アーキタイプを使用して、ルールを持つ DRL ファイルを作成し、アーキタイプを Red Hat Process Automation Manager デシジョンサービスに統合します。DRL ルールを作成する方法は、デシジョンサービスに外部 Maven アーキタイプを使用している場合や、同じワークフローを継続したい場合に便利です。この方法を使用しなくなった場合は、Red Hat Process Automation Manager の Business Central インターフェースを使用して、DRL ファイルや、その他のルールアセットを作成することが推奨されます。

手順

1. Maven アーキタイプを作成するディレクトリーに移動して、次のコマンドを実行します。

```
mvn archetype:generate -DgroupId=com.sample.app -DartifactId=my-app
-DarchetypeArtifactId=maven-archetype-quickstart -
DinteractiveMode=false
```

これにより、**my-app** という名前のディレクトリーが、以下の構造で作成されます。

```
my-app
|-- pom.xml
`-- src
    |-- main
    |   `-- java
    |       `-- com
    |           `-- sample
    |               `-- app
    |                   `-- App.java
    `-- test
        `-- java
            `-- com
                `-- sample
                    `-- app
                        `-- AppTest.java
```

my-app ディレクトリーには、以下の重要なコンポーネントが含まれます。

- アプリケーションソースを保存する **src/main** ディレクトリー
- テストソースを保存する **src/test** ディレクトリー
- プロジェクト設定ファイル **pom.xml**

2. Maven アーキタイプに、ルールが有効な Java オブジェクトを作成します。
この例では、**my-app/src/main/java/com/sample/app** ディレクトリーに **Person.java** ファイルが作成されます。**Person** クラスには、名前、苗字、時給、賃金を設定および取得するゲッターメソッドおよびセッターメソッドが含まれます。

```
package com.sample.app;
```

```
public class Person {

  private String firstName;
  private String lastName;
  private Integer hourlyRate;
  private Integer wage;

  public String getFirstName() {
    return firstName;
  }

  public void setFirstName(String firstName) {
    this.firstName = firstName;
  }

  public String getLastName() {
    return lastName;
  }

  public void setLastName(String lastName) {
    this.lastName = lastName;
  }

  public Integer getHourlyRate() {
    return hourlyRate;
  }

  public void setHourlyRate(Integer hourlyRate) {
    this.hourlyRate = hourlyRate;
  }

  public Integer getWage(){
    return wage;
  }

  public void setWage(Integer wage){
    this.wage = wage;
  }
}
```

3. **my-app/src/main/resources/rules** に、**.drl** 形式のルールファイルを作成します。DRL ファイルには、少なくともパッケージの指定と、(1 つまたは複数の) ルールで使用されるデータオブジェクトのインポートリストと、**when** 条件および **then** アクションを持つ 1 つ以上のルールが含まれます。
以下の **Wage.drl** ファイルには、**Person** クラスをインポートする **Wage** ルールが含まれ、賃金および時給の値を計算し、その結果に基づいてメッセージを表示します。

```
package com.sample.app;

import com.sample.app.Person;

dialect "java"

rule "Wage"
  when
    Person(hourlyRate * wage > 100)
```

```
    Person(name : firstName, surname : lastName)
  then
    System.out.println("Hello " + name + " " + surname + "!");
    System.out.println("You are rich!");
end
```

4. **my-app/src/main/resources/META-INF** ディレクトリーに、以下の内容の **kmodule.xml** メタデータファイルを作成します。

```
<?xml version="1.0" encoding="UTF-8"?>
<kmodule xmlns="http://www.drools.org/xsd/kmodule">
</kmodule>
```

この **kmodule.xml** ファイルは、KIE ベースへのリソースを選択し、セッションを設定する KIE モジュールの記述子です。このファイルを使用すると、KIE ベースを 1 つ以上定義して設定し、特定の KIE ベースの特定の **packages** から DRL ファイルを含めることができます。各 KIE ベースから KIE セッションを 1 つ以上作成することもできます。

次の例は、より高度な **kmodule.xml** ファイルを表示します。

```
<?xml version="1.0" encoding="UTF-8"?>
<kmodule xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://www.drools.org/xsd/kmodule">
  <kbase name="KBase1" default="true" eventProcessingMode="cloud"
equalsBehavior="equality" declarativeAgenda="enabled"
packages="org.domain.pkg1">
    <ksession name="KSession1_1" type="stateful" default="true" />
    <ksession name="KSession1_2" type="stateful" default="true"
beliefSystem="jtms" />
  </kbase>
  <kbase name="KBase2" default="false" eventProcessingMode="stream"
equalsBehavior="equality" declarativeAgenda="enabled"
packages="org.domain.pkg2, org.domain.pkg3" includes="KBase1">
    <ksession name="KSession2_1" type="stateless" default="true"
clockType="realtime">
      <fileLogger file="debugInfo" threaded="true" interval="10" />
      <workItemHandlers>
        <workItemHandler name="name" type="new
org.domain.WorkItemHandler()" />
      </workItemHandlers>
      <listeners>
        <ruleRuntimeEventListener
type="org.domain.RuleRuntimeListener" />
        <agendaEventListener type="org.domain.FirstAgendaListener"
/>
        <agendaEventListener type="org.domain.SecondAgendaListener"
/>
        <processEventListener type="org.domain.ProcessListener" />
      </listeners>
    </ksession>
  </kbase>
</kmodule>
```

この例は、KIE ベースを 2 つ定義します。KIE ベース **KBase1** から 2 つの KIE セッションをインスタンス化し、**KBase2** セッションから 1 つの KIE セッションをインスタンス化しま

す。**KBase2** の KIE セッションは、**ステートレス**な KIE セッションですが、これは 1 つ前の KIE セッションで呼び出されたデータ (1 つ前のセッションの状態) が、セッションの呼び出しと呼び出しの間で破棄されることを示しています。ルールアセットで特定の **packages** が両方の KIE ベースに含まれます。この方法でパッケージを指定した場合は、指定したパッケージを反映するフォルダー構造で DRL ファイルを整理する必要があります。

5. **my-app/pom.xml** 設定ファイルで、アプリケーションが要求するライブラリーを指定します。Red Hat Process Automation Manager の依存関係と、アプリケーションの **group ID**、**artifact ID**、および **version** (GAV) を提供します。

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
<modelVersion>4.0.0</modelVersion>
<groupId>com.sample.app</groupId>
<artifactId>my-app</artifactId>
<version>1.0.0</version>
<repositories>
  <repository>
    <id>jboss-ga-repository</id>
    <url>http://maven.repository.redhat.com/ga/</url>
  </repository>
</repositories>
<dependencies>
  <dependency>
    <groupId>org.drools</groupId>
    <artifactId>drools-compiler</artifactId>
    <version>VERSION</version>
  </dependency>
  <dependency>
    <groupId>org.kie</groupId>
    <artifactId>kie-api</artifactId>
    <version>VERSION</version>
  </dependency>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.11</version>
    <scope>test</scope>
  </dependency>
</dependencies>
</project>
```

Red Hat Process Automation Manager における Maven 依存関係および BOM (Bill of Materials) については、「What is the mapping between Red Hat Process Automation Manager and Maven library version?」を参照してください。

6. **my-app/src/test/java/com/sample/app/AppTest.java** の **testApp** メソッドを使用してルールをテストします。Maven によって、**AppTest.java** ファイルがデフォルトで作成されます。

7. **AppTest.java** ファイルで、KIE サービス、KIE コンテナー、および KIE セッションをインポートするのに必要な **import** 命令文を追加します。次に、KIE ベースをロードし、ファクトを挿入し、ファクトモデルをルールに渡す **testApp()** メソッドからルールを実行します。

```
import org.kie.api.KieServices;
import org.kie.api.runtime.KieContainer;
import org.kie.api.runtime.KieSession;

public void testApp() {

  // Load the KIE base:
  KieServices ks = KieServices.Factory.get();
  KieContainer kContainer = ks.getKieClasspathContainer();
  KieSession kSession = kContainer.newKieSession();

  // Set up the fact model:
  Person p = new Person();
  p.setWage(12);
  p.setFirstName("Tom");
  p.setLastName("Summers");
  p.setHourlyRate(10);

  // Insert the person into the session:
  kSession.insert(p);

  // Fire all rules:
  kSession.fireAllRules();
  kSession.dispose();
}
```

8. Maven アーキタイプにすべての DRL アセットを作成して保存したあと、コマンドラインで **my-app** ディレクトリーに移動し、以下のコマンドを実行してファイルを作成します。

```
mvn clean install
```

ビルドに失敗したら、コマンドラインのエラーメッセージに記載されている問題に対応し、ビルドに成功するまでファイルの妥当性確認を行います。

9. ファイルが問題なくビルドできたら、以下のコマンドを実行してローカルでルールを実行します。**com.sample.app** をパッケージ名に置き換えます。

```
mvn exec:java -Dexec.mainClass="com.sample.app"
```

10. ルールを見直して、適切に実行したことを確認し、ファイルで必要な変更に対応します。

Red Hat Process Automation Manager で既存のプロジェクトと新しいルールアセットを統合するには、ナレッジJAR (KJAR) として新規 Maven プロジェクトをコンパイルし、Business Central でプロジェクトの **pom.xml** ファイルに、依存関係としてこのプロジェクトを追加します。Business Central でプロジェクト **pom.xml** にアクセスするには、プロジェクトで既存のアセットを選択し、画面左側の **Project Explorer** メニューで **Customize View** ギアアイコンをクリックし、**Repository View** → **pom.xml** を選択します。

6.4. 実行可能ルールモデル

実行可能ルールモデルは埋め込み可能なモデルで、ビルド時に実行するルールセットの Java ベース表記を提供します。実行可能モデルは Red Hat Process Automation Manager の標準アセットパッケージングの代わりとなるもので、より効率的です。KIE コンテナーと KIE ベースの作成がより迅速にでき、DRL (Drools Rule Language) ファイルリストや他の Red Hat Process Automation Manager アセットが多い場合は、特に有効です。このモデルは詳細レベルにわたり、インデックス評価の lambda 表記など、必要な実行情報すべてを提供できます。

実行可能なルールモデルでは、プロジェクトにとって具体的に以下のような利点があります。

- **コンパイル時間:** 従来のパッケージ化された Red Hat Process Automation Manager プロジェクト (KJAR) には、制限や結果を実装する事前生成済みのクラスと合わせて、ルールベースを定義する DRL ファイルのリストやその他の Red Hat Process Automation Manager アーティファクトが含まれています。これらの DRL ファイルは、KJAR が Maven リポジトリーからダウンロードされて、KIE コンテナーにインストールされた時点で、解析してコンパイルする必要があります。特に大規模なルールセットの場合など、このプロセスは時間がかかる可能性があります。実行可能なモデルでは、プロジェクト KJAR 内で、Java クラスをパッケージして、プロジェクトルールベースの実行可能なモデルを実装し、はるかに早い方法で KIE コンテナーと KIE ベースをを再作成することができます。Maven プロジェクトでは、**`kie-maven-plugin`** を使用してコンパイルプロセス中に DRL ファイルから 実行可能なモデルソースを自動的に生成します。

- **ランタイム:** 実行可能なモデルでは、制約はすべて、Java lambda 式で定義されます。同じ lambda 式も制約評価に使用するので、**`mvel`** ベースの制約をバイトコードに変換するのに、解釈評価用の **`mvel`** 式も、Just-In-Time (JIT) プロセスも使用しません。これにより、よりすばやく効率的なランタイムを構築できます。

- **開発時間:** 実行可能なモデルでは、DRL 形式で直接要素をエンコードしたり、DRL パーサーを対応するように変更したりする必要なく、プロセスエンジンの新機能で開発および試行することができます。

6.4.1. Maven プロジェクトへの実行可能なルールモデルの埋め込み

Maven プロジェクトに実行可能ルールモデルを埋め込み、ビルド時にルールアセットをより効率的にコンパイルすることができます。

前提条件/事前作業

Red Hat Process Automation Manager ビジネスアセットを含む Maven 化したプロジェクトがある。

手順

1. Maven プロジェクトの **`pom.xml`** ファイルで、パッケージタイプを **`kjar`** に設定し、**`kie-maven-plugin`** ビルドコンポーネントを追加します。

```
<packaging>kjar</packaging>
...
<build>
  <plugins>
    <plugin>
      <groupId>org.kie</groupId>
      <artifactId>kie-maven-plugin</artifactId>
      <version>${rhpam.version}</version>
      <extensions>true</extensions>
    </plugin>
  </plugins>
</build>
```

kjar パッケージングタイプは、**kie-maven-plugin** コンポーネントをアクティブにして、アーティファクトリソースを検証してプリコンパイルします。**<version>** は、プロジェクトで現在使用される Red Hat Process Automation Manager の Maven アーティファクトのバージョン (例: 7.14.0.Final-redhat-00004) で、デプロイメントに Maven プロジェクトを適切にパッケージがするのに必要です。

注記

個別の依存関係に対して Red Hat Process Automation Manager **<version>** を指定するのではなく、Red Hat Business Automation 部品表 (BOM) の依存関係をプロジェクトの **pom.xml** ファイルに追加することを検討してください。Red Hat Business Automation BOM は、Red Hat Decision Manager と Red Hat Process Automation Manager の両方に適用します。BOM ファイルを追加すると、指定の Maven リポジトリーからの一時的な依存関係の内、正しいバージョンが、このプロジェクトに追加されます。

BOM 依存関係の例:

```
<dependency>
  <groupId>com.redhat.ba</groupId>
  <artifactId>ba-platform-bom</artifactId>
  <version>7.2.0.GA-redhat-00004</version>
  <scope>import</scope>
  <type>pom</type>
</dependency>
```

Red Hat Business Automation BOM (Bill of Materials) についての詳細情報は、What is the mapping between RHPAM product and maven library version? を参照してください。

2. 以下の依存関係を **pom.xml** ファイルに追加して、ルールアセットが実行可能なモデルからビルドできるようにします。

   - **drools-canonical-model**: Red Hat Process Automation Manager から独立するルールセットモデルの実行可能な正規表現を有効にします。
   - **drools-model-compiler**: デシジョンエンジンでRed Hat Process Automation Manager の内部データ構造に実行可能なモデルをコンパイルします。

```
<dependency>
  <groupId>org.drools</groupId>
  <artifactId>drools-canonical-model</artifactId>
  <version>${rhpam.version}</version>
</dependency>

<dependency>
  <groupId>org.drools</groupId>
  <artifactId>drools-model-compiler</artifactId>
  <version>${rhpam.version}</version>
</dependency>
```

3. コマンドターミナルで Maven プロジェクトディレクトリーに移動して、以下のコマンドを実行し、実行可能なモデルからプロジェクトをビルドします。

```
mvn clean install -DgenerateModel=<VALUE>
```

-DgenerateModel=<VALUE> プロパティーで、プロジェクトが DRL ベースの KJAR ではなく、モデルベースの KJAR としてビルドできるようにします。

<VALUE> は、3 つの値のいずれかに置き換えます。

- **YES**: オリジナルプロジェクトの DRL ファイルに対応する実行可能なモデルを生成し、生成した KJAR から DRL ファイルを除外します。
- **WITHDRL**: オリジナルプロジェクトの DRL ファイルに対応する実行可能なモデルを生成し、文書化の目的で、生成した KJAR に DRL ファイルを追加します (KIE ベースはいずれの場合でも実行可能なモデルからビルドされます)。
- **NO**: 実行可能なモデルは生成されません。

ビルドコマンドの例:

```
mvn clean install -DgenerateModel=YES
```

Maven プロジェクトのパッケージ化に関する詳細は、**「Packaging and deploying a Red Hat Process Automation Manager project」**を参照してください。

6.4.2. Java アプリケーションページへの実行可能なルールモデルの埋め込み

Java アプリケーションに実行可能ルールモデルをプログラミングを使用して埋め込み、ビルド時にルールアセットをより効率的にコンパイルすることができます。

前提条件/事前作業

Red Hat Process Automation Manager ビジネスアセットを含む Java アプリケーションがあること

手順

1. Java プロジェクトの適切なクラスパスに、以下の依存関係を追加します。
   - **drools-canonical-model**: Red Hat Process Automation Manager から独立するルールセットモデルの実行可能な正規表現を有効にします。
   - **drools-model-compiler**: デシジョンエンジンでRed Hat Process Automation Manager の内部データ構造に実行可能なモデルをコンパイルします。

```
<dependency>
  <groupId>org.drools</groupId>
  <artifactId>drools-canonical-model</artifactId>
  <version>${rhpam.version}</version>
</dependency>

<dependency>
  <groupId>org.drools</groupId>
  <artifactId>drools-model-compiler</artifactId>
  <version>${rhpam.version}</version>
</dependency>
```

<version> は、プロジェクトで現在使用する Red Hat Process Automation Manager の Maven アーティファクトバージョンです (例: 7.14.0.Final-redhat-00004)。

注記

個別の依存関係に対して Red Hat Process Automation Manager **<version>** を指定するのではなく、Red Hat Business Automation 部品表 (BOM) の依存関係をプロジェクトの **pom.xml** ファイルに追加することを検討してください。Red Hat Business Automation BOM は、Red Hat Decision Manager と Red Hat Process Automation Manager の両方に適用します。BOM ファイルを追加すると、指定の Maven リポジトリーからの一時的な依存関係の内、正しいバージョンが、このプロジェクトに追加されます。

BOM 依存関係の例:

```
<dependency>
  <groupId>com.redhat.ba</groupId>
  <artifactId>ba-platform-bom</artifactId>
  <version>7.2.0.GA-redhat-00004</version>
  <scope>import</scope>
  <type>pom</type>
</dependency>
```

Red Hat Business Automation BOM (Bill of Materials) についての詳細情報は、What is the mapping between RHPAM product and maven library version? を参照してください。

2. ルールアセットを KIE 仮想ファイルシステム **KieFileSystem** に追加して、**KieBuilder** に **buildAll(ExecutableModelProject.class)** を指定して使用し、実行可能なモデルからアセットをビルドします。

```
import org.kie.api.KieServices;
import org.kie.api.builder.KieFileSystem;
import org.kie.api.builder.KieBuilder;

  KieServices ks = KieServices.Factory.get();
  KieFileSystem kfs = ks.newKieFileSystem()
  kfs.write("src/main/resources/KBase1/ruleSet1.drl",
stringContainingAValidDRL)
  .write("src/main/resources/dtable.xls",

kieServices.getResources().newInputStreamResource(dtableFileStream))
;

  KieBuilder kieBuilder = ks.newKieBuilder( kfs );
  // Build from an executable model
  kieBuilder.buildAll( ExecutableModelProject.class )
  assertEquals(0,
kieBuilder.getResults().getMessages(Message.Level.ERROR).size());
```

実行可能なモデルから **KieFileSystem** をビルドした後に、作成された **KieSession** は効率のあまりよくない **mvel** 式ではなく、lambda 四季をもとにした制約を使用します。**buildAll()** に引数が含まれていない場合には、プロジェクトは実行可能なモデルのない標準の手法でビルドされます。

KieFileSystem を使用する代わりに、手作業を多く使用して実行可能なモデルを作成する別の方法として、Fluent API で **Model** を定義して、そこから **KieBase** を作成することができます。

```
Model model = new ModelImpl().addRule( rule );
KieBase kieBase = KieBaseBuilder.createKieBaseFromModel( model );
```

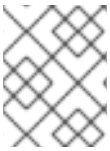
Java アプリケーション内でプロジェクトをプログラミングを使用してパッケージ化する方法については、『**Packaging and deploying a Red Hat Process Automation Manager project**』を参照してください。

第7章 RED HAT PROCESS AUTOMATION MANAGER の IDE 向けのデシジョン例

Red Hat Process Automation Manager は、統合開発環境 (IDE: integrated development environment) にインポートできるように Java クラスとして配信される、デシジョン例を提供します。これらの例は、Red Hat Process Automation Manager のデシジョンエンジン機能をさらに理解するために使用するか、Red Hat Process Automation Manager プロジェクトに定義するデシジョンの参考として使用してください。

以下のデシジョンセットの例は、Red Hat Process Automation Manager で利用可能な例の一部です。

- **Hello World の例**: 基本的なルール実行や、デバッグ出力の使用方法を例示します。
- **状態の例**: ルールの顕著性やアジェンダグループを使用した前向き連鎖や競合解決を例示します。
- **フィボナッチの例**: ルールの顕著性を使用した再帰や競合解決を例示します。
- **銀行の例**: パターン一致、基本的なソート、計算を例示します。
- **ペットショップの例**: ルールアジェンダグループ、グローバル変数、コールバック、GUI 統合を例示します。
- **数独の例**: 複雑なパターン一致、問題解決、コールバック、GUI 統合を例示します。
- **House of Doom の例**: 後向き連鎖と再帰を例示します。



注記

Red Hat Business Optimizer が提供する最適化の例については、**「Getting started with Red Hat Business Optimizer」**を参照してください。

7.1. IDE での RED HAT PROCESS AUTOMATION MANAGER のデシジョン例のインポートと実行

Red Hat Process Automation Manager のデシジョン例を統合開発環境 (IDE) にインポートして実行し、ルールとコードがどのように機能するかチェックできます。これらの例は、Red Hat Process Automation Manager のデシジョンエンジン機能をさらに理解するために使用するか、Red Hat Process Automation Manager プロジェクトに定義するデシジョンの参考として使用してください。

前提条件

- Java 8 以降をインストールしていること
- Maven 3.5.x 以降をインストールしていること
- Red Hat JBoss Developer Studio など IDE がインストールされていること

手順

1. Red Hat カスタマーポータル から **Red Hat Process Automation Manager 7.2.0 Source Distribution** を `/rhpam-7.2.0-sources` など、一時的なディレクトリーに、ダウンロードして展開します。

2. IDE を開き、**File** → **Import** → **Maven** → **Existing Maven Projects** を選択するか、同等のオプションを選択して、Maven プロジェクトをインポートします。

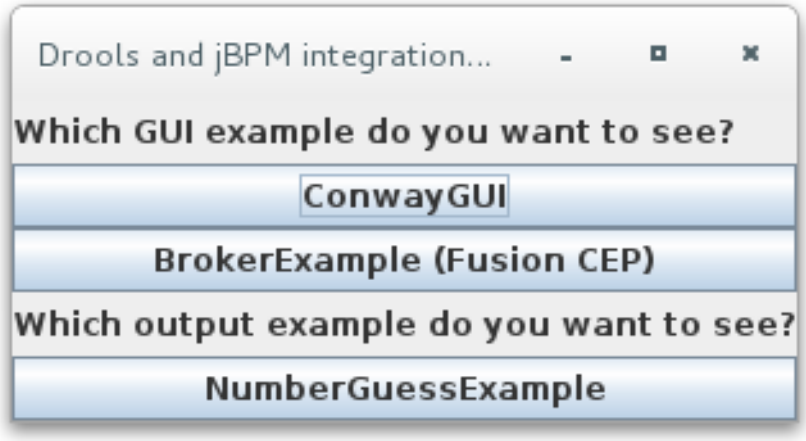
3. **Browse** をクリックして、**`~/rhpam-7.2.0-sources/src/drools-$VERSION/drools-examples`** (または、Conway の Game of Life の例の場合は、**`~/rhpam-7.2.0-sources/src/droolsjbpm-integration-$VERSION/droolsjbpm-integration-examples`**) に移動して、プロジェクトをインポートします。

4. 実行するパッケージ例に移動して、**`main`** メソッドが含まれる Java クラスを検索します。

5. Java クラスを右クリックし、**Run As** → **Java Application** を選択して例を実行します。
基本的なユーザーインターフェースですべての例を実行するには、**`org.drools.examples`** Main クラスの **`DroolsExamplesApp.java`** クラス (または Conway の Game of Life の場合は **`DroolsJbpmIntegrationExamplesApp.java`** クラス) を実行します。

図7.1 drools-examples (DroolsExamplesApp.java) 内のすべての例のインターフェース

図7.2 droolsjbpm-integration-examples (DroolsJbpmIntegrationExamplesApp.java) のすべての例のインターフェース



7.2. HELLO WORLD の例のデシジョン (基本ルールおよびデバッグ)

Hello World のデシジョン例セットは、オブジェクトを Red Hat Process Automation Manager デシジョンエンジンの作業メモリーに挿入する方法、ルールを使用してオブジェクトを照合する方法、エンジンの内部アクティビティーを追跡するロギングの設定方法を例示します。

以下は、Hello World の例の概要です。

- **名前**: **helloworld**
- **Main クラス**: (**src/main/java** 内の) **org.drools.examples.helloworld.HelloWorldExample**
- **モジュール**: **drools-examples**
- **タイプ**: Java アプリケーション
- **ルールファイル**: (**src/main/resources** 内の) **org.drools.examples.helloworld.HelloWorld.drl**
- **目的**: 基本的なルール実行とデバッグ出力の使用方法を例示します。

Hello World の例では、KIE セッションが生成されて、ルールの実行が可能になります。すべてのルールは、実行できるように KIE セッションが必要です。

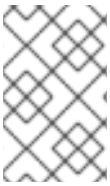
ルール実行の KIE セッション

```
KieServices ks = KieServices.Factory.get(); (1)
KieContainer kc = ks.getKieClasspathContainer(); (2)
KieSession ksession = kc.newKieSession("HelloWorldKS"); (3)
```

(1) **KieServices** ファクトリーを取得します。これは、アプリケーションがエンジンとの対話に使用する主なインターフェースです。

(2) プロジェクトクラスパスから **KieContainer** を作成します。これで、**/META-INF/kmodule.xml** ファイルを検出し、このファイルをもとに設定して **KieModule** で **KieContainer** をインスタンス化します。

3 **/META-INF/kmodule.xml** ファイルに定義された **"HelloWorldKS"** KIE セッション設定をもとに **KieSession** を作成します。



注記

Red Hat Process Automation Manager プロジェクトのパッケージ化に関する詳細は、**『Packaging and deploying a Red Hat Process Automation Manager project』**を参照してください。

Red Hat Process Automation Manager には、内部エンジンアクティビティーを公開するイベントモデルがあります。**DebugAgendaEventListener** と **DebugWorkingMemoryEventListener** のデフォルトのデバッグリスナー 2 つにより、デバッグイベント情報が **System.err** の出力に表示されます。**KieRuntimeLogger** では、実行監査と、グラフィックビューワーで確認可能な結果が提供されます。

リスナーと監査ロガーのデバッグ

```
// Set up listeners.
ksession.addEventListener( new DebugAgendaEventListener() );
ksession.addEventListener( new DebugRuleRuntimeEventListener() );

// Set up a file-based audit logger.
KieRuntimeLogger logger = KieServices.get().getLoggers().newFileLogger(
ksession, "./target/helloworld" );

// Set up a ThreadedFileLogger so that the audit view reflects events
while debugging.
KieRuntimeLogger logger = ks.getLoggers().newThreadedFileLogger( ksession,
"./target/helloworld", 1000 );
```

ロガーは、**Agenda** と **RuleRuntime** リスナーにビルドされる特別な実装です。エンジンが実行を終えると、**logger.close()** が呼び出されます。

この例では、**"Hello World"** というメッセージを含む **Message** オブジェクトを作成し、ステータス **HELLO** を **KieSession** に挿入して、**fireAllRules()** でルールを実行します。

データの挿入および実行

```
// Insert facts into the KIE session.
final Message message = new Message();
message.setMessage( "Hello World" );
message.setStatus( Message.HELLO );
ksession.insert( message );

// Fire the rules.
ksession.fireAllRules();
```

ルール実行は、データモデルを使用して、**KieSession** への出入力としてデータを渡します。この例のデータモデルには **message** (**String**) と **status** (**HELLO** または **GOODBYE**) の2 つのフィールドが含まれます。

データモデルクラス

-

```
public static class Message {
    public static final int HELLO   = 0;
    public static final int GOODBYE = 1;

    private String          message;
    private int             status;
    ...
}
```

この 2 つのルールは、**src/main/resources/org/drools/examples/helloworld/HelloWorld.drl** ファイルに配置されます。

"Hello World" ルールの **when** 条件では、ステータスが **Message.HELLO** の KIE セッションに、**Message** オブジェクトが挿入されるたびに、このルールをアクティベートすると記述しています。さらに、変数のバインドが 2 つ作成されます (**message** 変数を **message** 属性に、**m** 変数を一致する **Message** オブジェクト自体にバインド)。

ルールの **then** アクションは、ルールの **dialect** 属性に宣言されているように、MVEL 式言語を使用して記述されます。**message** の束縛変数のコンテンツを **System.out** に出力した後に、ルールは **m** にバインドされている **Message** オブジェクトの **message** と **status** 属性値を変更します。このルールは MVEL の **modify** ステートメントを使用して、1 つのステートメントに割り当てブロックを適用し、この変更についてブロックの最後にエンジンに通知します。

"Hello World" のルール

```
rule "Hello World"
      dialect "mvel"
  when
    m : Message( status == Message.HELLO, message : message )
  then
    System.out.println( message );
    modify ( m ) { message = "Goodbye cruel world",
                   status = Message.GOODBYE };
end
```

java 方言を指定する **"Good Bye"** ルールは、ステータスが **Message.GOODBYE** の **Message** オブジェクトと一致する点を除き、**"Hello World"** ルールによく似ています。

"Good Bye" ルール

```
rule "Good Bye"
      dialect "java"
  when
    Message( status == Message.GOODBYE, message : message )
  then
    System.out.println( message );
end
```

この例を実行するには、**org.drools.examples.helloworld.HelloWorldExample** クラスを IDE で Java アプリケーションとして実行します。このルールは **System.out** に、デバッグリスナーは **System.err** に書き込み、監査ロガーは **target/helloworld.log** のログファイルを作成します。

IDE コンソールの System.out 出力

```
Hello World
Goodbye cruel world
```

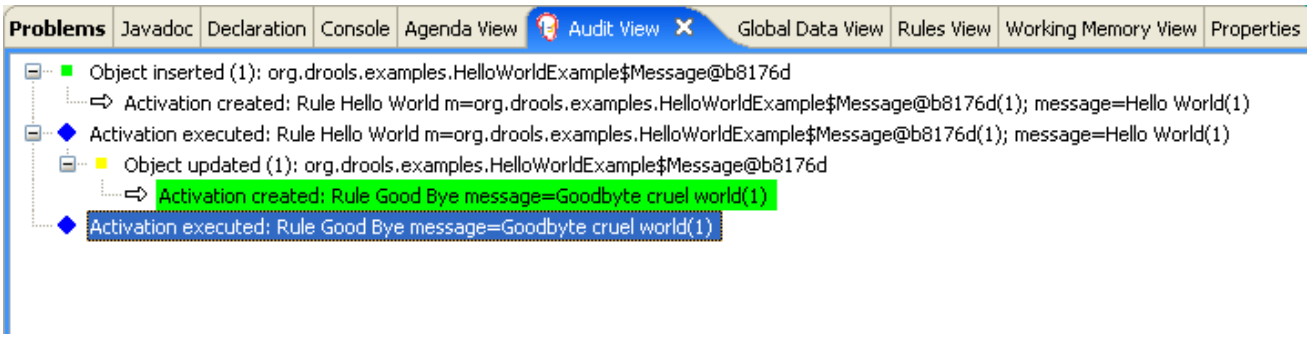
IDE コンソールでの System.err の出力

```
==>[ActivationCreated(0): rule=Hello World;
                    tuple=
[fid:1:1:org.drools.examples.helloworld.HelloWorldExample$Message@17cec96]
]
[ObjectInserted: handle=
[fid:1:1:org.drools.examples.helloworld.HelloWorldExample$Message@17cec96]
;

object=org.drools.examples.helloworld.HelloWorldExample$Message@17cec96]
[BeforeActivationFired: rule=Hello World;
                    tuple=
[fid:1:1:org.drools.examples.helloworld.HelloWorldExample$Message@17cec96]
]
==>[ActivationCreated(4): rule=Good Bye;
                    tuple=
[fid:1:2:org.drools.examples.helloworld.HelloWorldExample$Message@17cec96]
]
[ObjectUpdated: handle=
[fid:1:2:org.drools.examples.helloworld.HelloWorldExample$Message@17cec96]
;

old_object=org.drools.examples.helloworld.HelloWorldExample$Message@17cec9
6;

new_object=org.drools.examples.helloworld.HelloWorldExample$Message@17cec9
6]
[AfterActivationFired(0): rule=Hello World]
[BeforeActivationFired: rule=Good Bye;
                    tuple=
[fid:1:2:org.drools.examples.helloworld.HelloWorldExample$Message@17cec96]
]
[AfterActivationFired(4): rule=Good Bye]
```

この例の実行フローをさらに理解するには、**target/helloworld.log** からの監査ログファイルを IDE デバッグビューまたは **Audit View** が利用できる場合は Audit View (例: IDE の **Window** → **Show View**) にロードします。

この例では、**Audit view** で、オブジェクトが挿入され、**"Hello World"** ルールのアクティベーションが作成されます。次に、このアクティベーションが実行され、**Message** オブジェクトを更新して、**"Good Bye"** ルールのアクティベーションをトリガーします。最後に、**"Good Bye"** ルールが実行されます。**Audit View** でインベントが選択されると、この例の **"Activation created"** イベントである元のイベントが緑色にハイライトされます。

図7.3 Hello World の例の監査ビュー



7.3. 状態の例のデシジョン (前向き連鎖および競合解決)

状態の例のディジョンセットでは、デシジョンエンジンが前向き連鎖と、作業メモリー内のファクトへの変更をどのように使用してルールの実行競合を順番に解決していくのかを例示します。この例では、ルールで定義可能な顕著性の値またはアジェンダグループを使用して競合を解決することにフォーカスします。

以下は、状態の例の概要です。

- **名前**: **`state`**
- **Main クラス**: (**`src/main/java`** 内の) **`org.drools.examples.state.StateExampleUsingSalience`**、**`org.drools.examples.state.StateExampleUsingAgendaGroup`**
- **モジュール**: **`drools-examples`**
- **タイプ**: Java アプリケーション
- **ルールファイル**: (**`src/main/resources`** 内の) **`org.drools.examples.state.*.drl`**
- **目的**: ルールの顕著性やアジェンダグループを使用した前向き連鎖や競合解決を例示します。

前向き連鎖のルールシステムは、ディジョンエンジンの作業メモリーにあるファクトで開始して、そのファクトへの変更に反応するデータ駆動型のシステムです。オブジェクトが作業メモリーに挿入されると、その変更の結果として True となってルールの条件が、アジェンダにより実行がスケジュールされます。

反対に、後向き連鎖のルールシステムは、通常再帰を使用して、デシジョンエンジンが満たそうとする結論から開始する目的駆動型のシステムです。システムが結論または目的に到達できない場合には、サブとなる目的、つまり、現在の目的の一部を完了する結論を検索します。システムは、最初の結論が満たされるか、すべてのサブとなる目的が満たされるまで続行されます。

状態の例では、**`State`** クラスごとに、名前や現在の状態のフィールドが含まれます (**`org.drools.examples.state.State`** のクラス参照)。以下の状態は、各プロジェクトで考えられる状態 2 つです。

- **`NOTRUN`**
- **`FINISHED`**

State クラス

```
public class State {
```

```
    public static final int NOTRUN   = 0;
    public static final int FINISHED = 1;

    private final PropertyChangeSupport changes =
        new PropertyChangeSupport( this );

    private String name;
    private int    state;

    ... setters and getters go here...
}
```

状態の例には、同じ例が 2 つのバージョンとして提供されており、それぞれルール実行の競合を解決します。

- ルールの顕著性を使用して競合を解決する **StateExampleUsingSalience** バージョン
- ルールアジェンダグループを使用して競合を解決する **StateExampleUsingAgendaGroups** バージョン

状態の例のバージョンはいずれも、**A**、**B**、**C**、**D** の 4 つの **State** オブジェクトを使用します。最初に、それぞれの状態は、**NOTRUN** に設定されます。NOTRUN は、例が使用するコンストラクターのデフォルト値です。

顕著性を使用した状態の例

状態の例の **StateExampleUsingSalience** バージョンでは、ルールで顕著性の値を使用し、ルール実行の競合を解決します。顕著性の値が高いルールは、アクティベーションキューの順番で、優先度が高くなります。

この例では、各 **State** インスタンスを KIE セッションに挿入して、**fireAllRules()** を呼び出します。

顕著性の状態例の実行

```
final State a = new State( "A" );
final State b = new State( "B" );
final State c = new State( "C" );
final State d = new State( "D" );

ksession.insert( a );
ksession.insert( b );
ksession.insert( c );
ksession.insert( d );

ksession.fireAllRules();

// Dispose KIE session if stateful (not required if stateless).
ksession.dispose();
```

この例を実行するには、IDE で Java アプリケーションとして **org.drools.examples.state.StateExampleUsingSalience** クラスを実行します。

実行後に、以下の出力が IDE コンソールウィンドウに表示されます。

IDE コンソールでの顕著性の状態例の出力

```
A finished
B finished
C finished
D finished
```

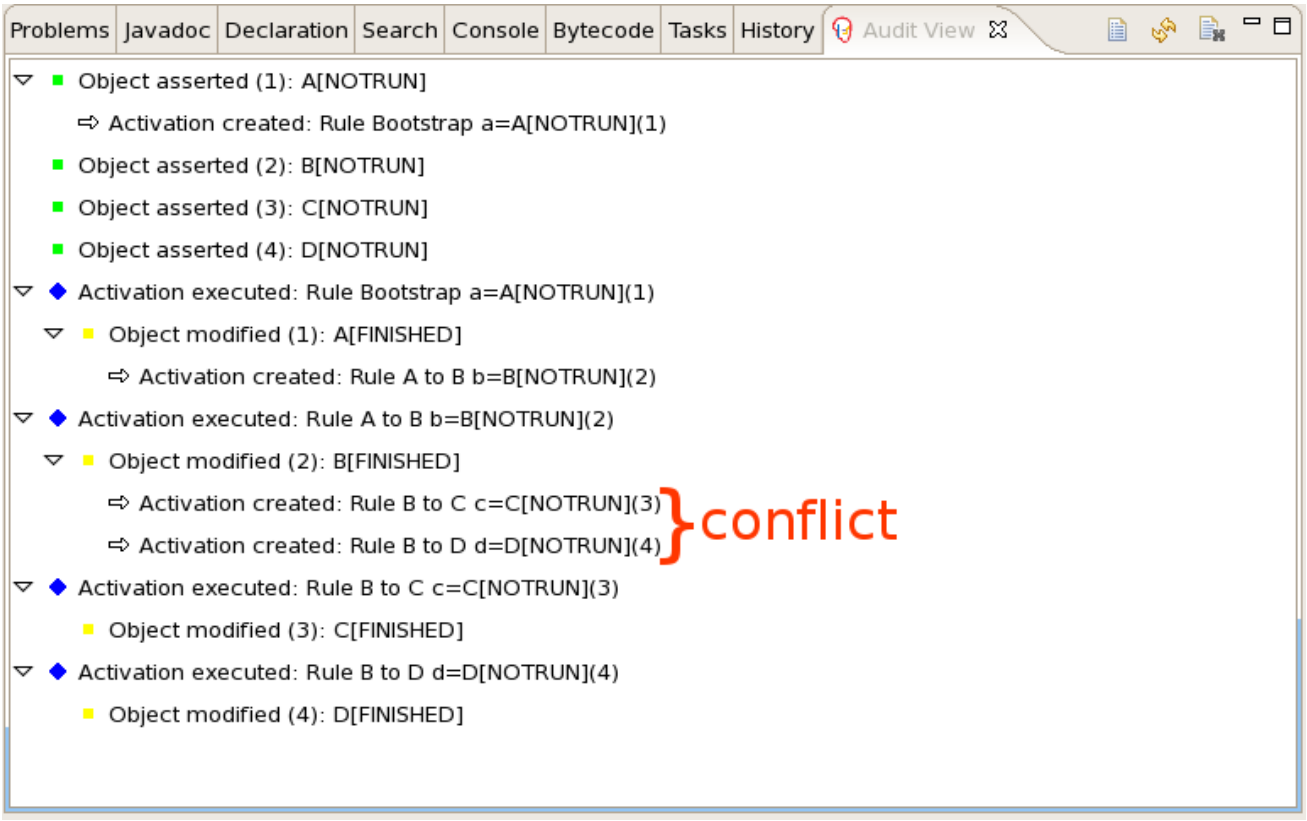
4 つのルールが存在します。

まず、**"Bootstrap"** ルールが実行され、**A** の状態が **FINISHED** に設定されます。次に、**B** の状態が **FINISHED** に変更され、オブジェクト **C** と **D** はいずれも **B** に依存するため、競合が発生しますが、顕著性の値で解決されます。

この例の実行フローをさらに理解するには、**target/state.log** からの監査ログファイルを IDE デバッグビューまたは **Audit View** が利用できる場合は Audit View (例: IDE の **Window** → **Show View**) にロードします。

この例では、**Audit View** は、状態が **NOTRUN** のオブジェクト **A** のアサーションが **"Bootstrap"** ルールをアクティベートしますが、他のオブジェクトのアサーションはすぐに有効になりません。

図7.4 顕著性の状態例の監査ビュー



顕著性の状態例の "Bootstrap" ルール

```
rule "Bootstrap"
  when
    a : State(name == "A", state == State.NOTRUN )
  then
    System.out.println(a.getName() + " finished" );
    a.setState( State.FINISHED );
end
```

"Bootstrap" ルールを実行すると、**A** の状態が **FINISHED** に変わり、ルール **"A to B"** をアクティベートします。

顕著性の状態例の "A to B" ルール

```
rule "A to B"
  when
    State(name == "A", state == State.FINISHED )
    b : State(name == "B", state == State.NOTRUN )
  then
    System.out.println(b.getName() + " finished" );
    b.setState( State.FINISHED );
end
```

"A to B" ルールを実行すると、**B** の状態を **FINISHED** に変更し、**"B to C"** と **"B to D"** 両方のルールをアクティベートして、これらのアクティベーションをエンジンアジェンダに配置します。

顕著性の状態例の "B to C" および "B to D" ルール

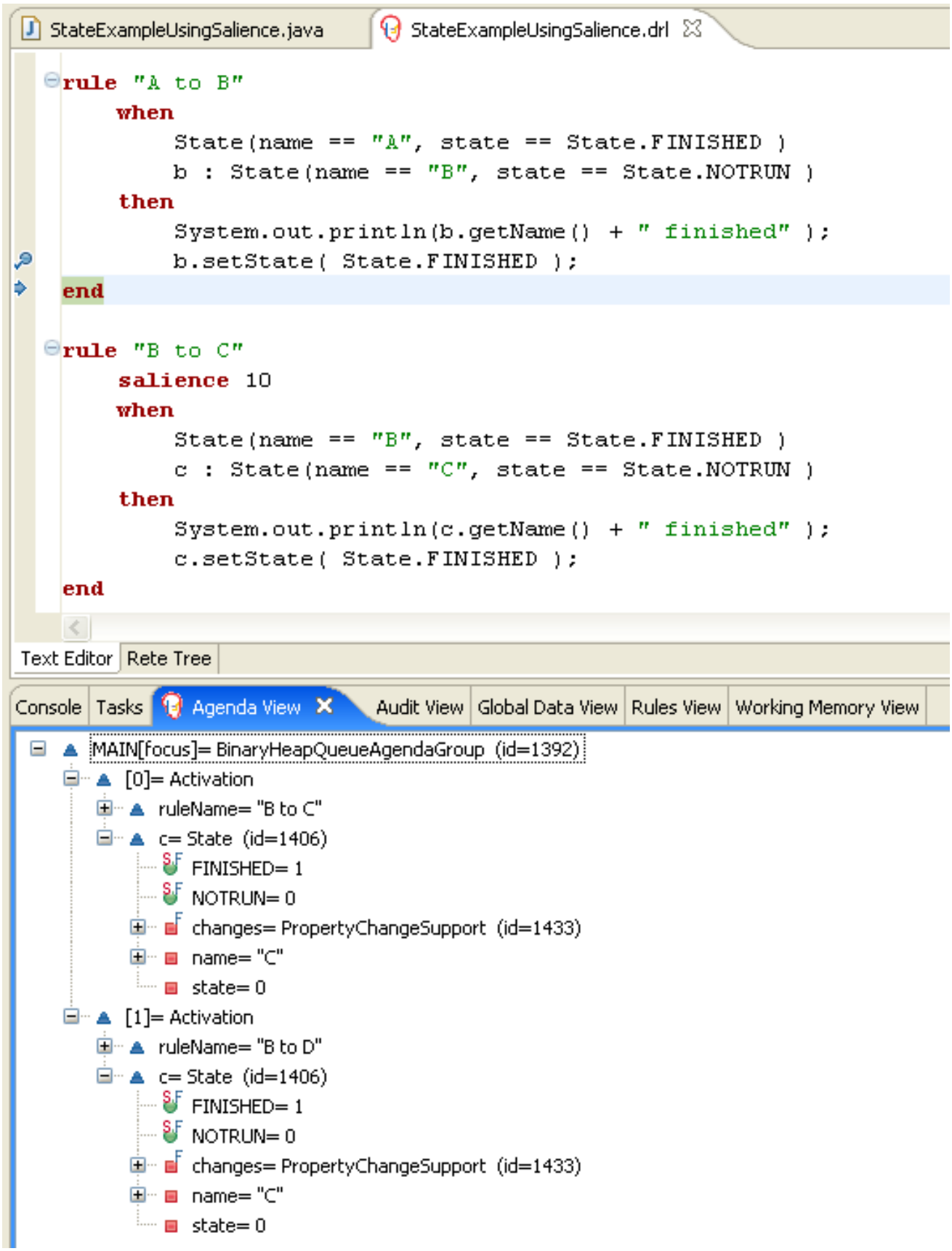
```
rule "B to C"
    salience 10
  when
    State(name == "B", state == State.FINISHED )
    c : State(name == "C", state == State.NOTRUN )
  then
    System.out.println(c.getName() + " finished" );
    c.setState( State.FINISHED );
end

rule "B to D"
  when
    State(name == "B", state == State.FINISHED )
    d : State(name == "D", state == State.NOTRUN )
  then
    System.out.println(d.getName() + " finished" );
    d.setState( State.FINISHED );
end
```

この時点から、両方のルールが実行される可能性があるため、これらのルールは競合しています。競合解決ストラテジーを使用すると、エンジンアジェンダがどのルールを実行するかを決定できます。**"B to C"** は、顕著性の値が高い (**10** と、デフォルトの顕著性の値 **0**) ので、先に実行され、オブジェクト **C** の状態が **FINISHED** に変更されます。

IDE の **Audit View** では、ルール **"A to B"** の **State** オブジェクトが変更され、2 つのアクティベーションが競合する結果になることが分かります。

IDE で **Agenda View** を使用して、エンジンアジェンダの状態を調査できます。この例では **Agenda View** で、ルール **"A to B"** のブレークポイントと、2 つの競合するルールを持つアジェンダの状態が分かります。最後にルール **"B to D"** が実行され、オブジェクト **D** の状態が **FINISHED** に変更されます。

図7.5 顕著性の状態例のアジェンダビュー



アジェンダグループを使用した状態の例

状態の例の **StateExampleUsingAgendaGroups** バージョンでは、ルールでアジェンダグループを使用し、ルール実行における競合を解決します。アジェンダグループを使用すると、エンジンアジェンダが分割され、ルールのグループの実行に対してこれまで以上に制御ができるようになります。デフォルトでは、ルールはすべてアジェンダグループ **MAIN** に含まれています。**agenda-group** 属性を使用して、ルールに異なるアジェンダグループを指定できます。

最初は、作業メモリーは、アジェンダグループ **MAIN** にフォーカスを当てます。アジェンダグループのルールは、グループがこのフォーカスを受けた場合のみ実行されます。**setFocus()** のメソッドか、**auto-focus** のルール属性を使用してフォーカスを設定できます。**auto-focus** 属性を使用すると、ルールが一致してアクティベートされた場合のみ、ルールにアジェンダグループのフォーカスが自動的に当てられます。

この例では、**auto-focus** 属性を使用すると **"B to D"** の前に **"B to C"** ルールを実行できます。

アジェンダグループの状態例のルール "B to C"

```
rule "B to C"
    agenda-group "B to C"
    auto-focus true
  when
    State(name == "B", state == State.FINISHED )
    c : State(name == "C", state == State.NOTRUN )
  then
    System.out.println(c.getName() + " finished" );
    c.setState( State.FINISHED );
    kcontext.getKnowledgeRuntime().getAgenda().getAgendaGroup( "B to D"
).setFocus();
end
```

ルール **"B to C"** は、アジェンダグループ **"B to D"** の **setFocus()** を呼び出し、アクティブなルールを実行できるようにします。その後にルール **"B to D"** が実行できるようになります。

アジェンダグループの状態例のルール "B to D"

```
rule "B to D"
    agenda-group "B to D"
  when
    State(name == "B", state == State.FINISHED )
    d : State(name == "D", state == State.NOTRUN )
  then
    System.out.println(d.getName() + " finished" );
    d.setState( State.FINISHED );
end
```

この例を実行するには、IDE で Java アプリケーションとして **org.drools.examples.state.StateExampleUsingAgendaGroups** クラスを実行します。

実行後に、以下の出力が IDE コンソールウィンドウに表示されます (状態の例の顕著性バージョンと同じ)。

IDE コンソールでのアジェンダグループの状態例の出力

```
A finished
B finished
C finished
D finished
```

状態の例の含まれる動的なファクト

状態の例に含まれる主なコンセプトとして他には、**PropertyChangeListener** オブジェクトを実装するオブジェクトをもとに **動的ファクト** を使用するというものがあります。エンジンがファクトプロ

パティーへの変更を確認し、対応するためには、アプリケーションがエンジンに対して、変更があったことを通知する必要があります。**modify** ステートメントを使用して、このコミュニケーションをルールで明示的に設定するか、JavaBeans 使用で定義されているようにファクトが **PropertyChangeSupport** インターフェースを実装するように指定することで暗黙的に設定できます。

この例は、ルールで **modify** ステートメントを明示的に指定しなくても良いように **PropertyChangeSupport** インターフェースを使用する方法が示されています。このインターフェースを使用するには、**org.drools.example.State** クラスと同じ方法で、ファクトに **PropertyChangeSupport** が実装されていることを確認し、DRL ルールファイルで以下のコードを使用して、これらのファクトでプロパティー変更がないかをリッスンするようにエンジンを設定してください。

動的ファクトの宣言

```
declare type State
  @propertyChangeSupport
end
```

PropertyChangeListener オブジェクトを使用する場合に、各セッターは通知用に追加のコードを実装する必要があります。たとえば、**state** の以下のセッターは **org.drools.examples** のクラスに含まれます。

PropertyChangeSupport のセッター例

```
public void setState(final int newState) {
    int oldState = this.state;
    this.state = newState;
    this.changes.firePropertyChange( "state",
                                     oldState,
                                     newState );
}
```

7.4. フィボナッチの例のデシジョン (再帰および競合解決)

フィボナッチの例のディジョンセットでは、デシジョンエンジンが再帰をどのように使用してルールの実行競合を順番に解決していくのかを例示します。この例では、ルールで定義可能な顕著性の値を使用して競合を解決することにフォーカスします。

以下は、フィボナッチの例の概要です。

- **名前**: **フィボナッチ**
- **Main クラス**: (**src/main/java** 内の) **org.drools.examples.fibonacci.FibonacciExample**
- **モジュール**: **drools-examples**
- **タイプ**: Java アプリケーション
- **ルールファイル**: (**src/main/resources** 内の) **org.drools.examples.fibonacci.Fibonacci.drl**
- **目的**: ルールの顕著性を使用した再帰や競合解決を例示します。

フィボナッチ数は、0 または 1 で開始する数列です。0、1、1、2、3、5、8、13、21、34、55、89、144、233、377、610、987、1597、2584、4181、6765、10946 などのように、2 つの先行する数を足すことにより、次にくるフィボナッチ数が求められます。

フィボナッチの例では、**Fibonacci** のファクトクラスを 1 つ使用し、このクラスに以下のフィールド 2 つが含まれています。

- **sequence**
- **value**

sequence フィールドは、フィボナッチ数列のオブジェクトの位置を示します。**value** フィールドは、その数列の位置のフィボナッチオブジェクトの値を示します。**-1** は、計算する必要がある値という意味です。

フィボナッチクラス

```
public static class Fibonacci {
    private int  sequence;
    private long value;

    public Fibonacci( final int sequence ) {
        this.sequence = sequence;
        this.value = -1;
    }

    ... setters and getters go here...
}
```

この例を実行するには、IDE で Java アプリケーションとして **org.drools.examples.fibonacci.FibonacciExample** クラスを実行します。

実行後に、以下の出力が IDE コンソールウィンドウに表示されます。

IDE コンソールでのフィボナッチの例の出力

```
recurse for 50
recurse for 49
recurse for 48
recurse for 47
...
recurse for 5
recurse for 4
recurse for 3
recurse for 2
1 == 1
2 == 1
3 == 2
4 == 3
5 == 5
6 == 8
...
47 == 2971215073
```

```
48 == 4807526976
49 == 7778742049
50 == 12586269025
```

Java でこの動作を実現するには、sequence フィールドに **50** を指定して、**Fibonacci** オブジェクトを挿入します。この例では、次に再帰ルールを使用して、他の 49 個の **Fibonacci** オブジェクトを挿入します。

PropertyChangeSupport インターフェースを実装して動的ファクトを使用する代わりに、このレでは MVEL 方言の **modify** キーワードを使用して、ブロックセッターアクションを有効にしてエンジンに変更を通知しています。

フィボナッチの例の実行

```
ksession.insert( new Fibonacci( 50 ) );
ksession.fireAllRules();
```

この例では、以下の 3 つのルールを使用します。

- **"Recurse"**
- **"Bootstrap"**
- **"Calculate"**

"Recurse" ルールは、値が **-1** の、アサートされた各 **Fibonacci** オブジェクトを照合して、現在の値よりも数列が 1 つ小さい **Fibonacci** オブジェクトを新たに作成し、アサートします。数列フィールドが **1** に相当するオブジェクトが存在しない場合に、フィボナッチオブジェクトが追加されると毎回、このルールは再度照合、実行されます。メモリーにフィボナッチオブジェクト 50 個すべてが存在する場合には、**not** 条件要素を使用して、ルールの合致を停止します。また、**"Bootstrap"** ルールを実行する前に **Fibonacci** オブジェクト 50 個すべてをアサートする必要があるので、このルールには **salience** の値も含まれます。

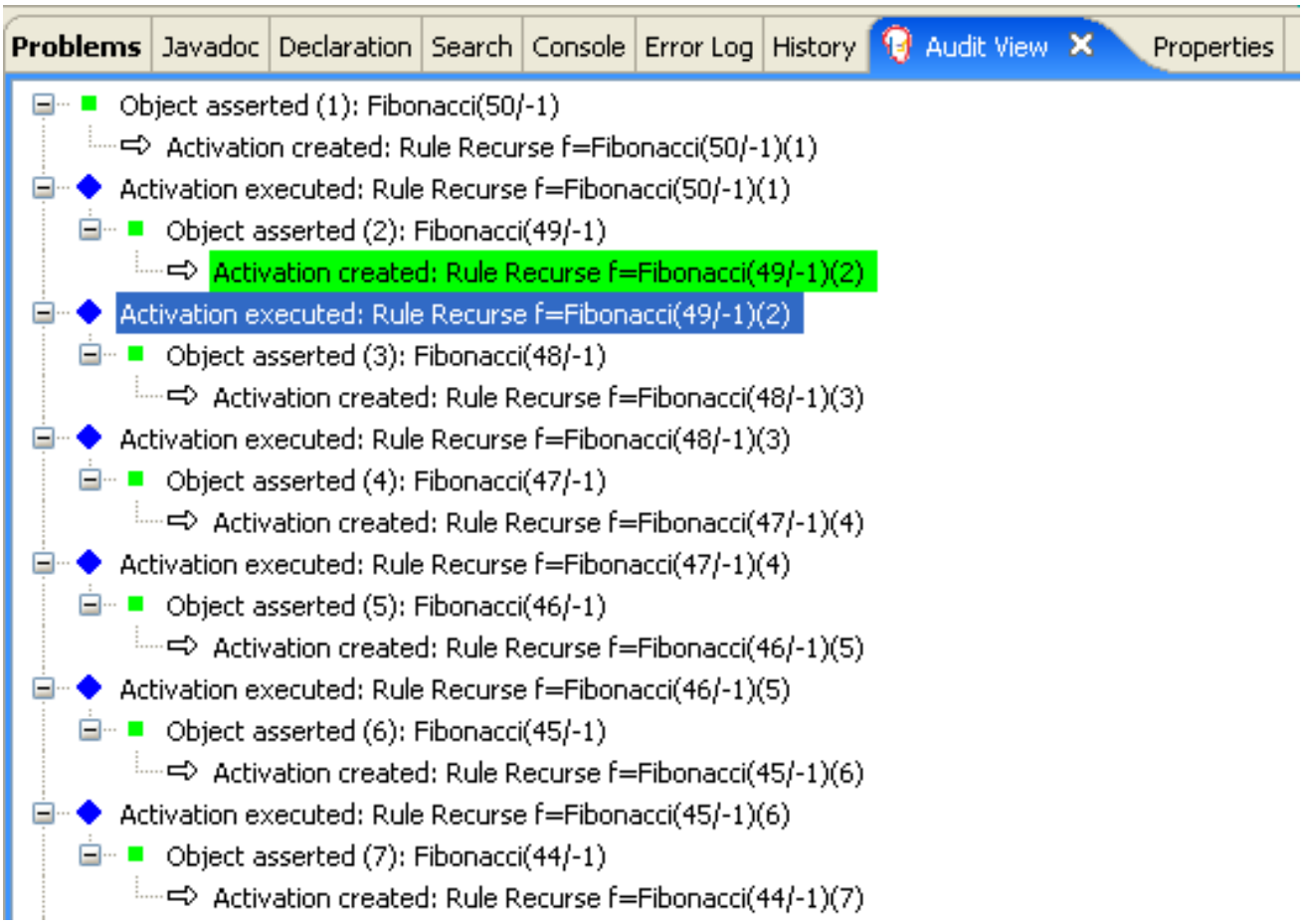
ルール "Recurse"

```
rule "Recurse"
    salience 10
  when
    f : Fibonacci ( value == -1 )
    not ( Fibonacci ( sequence == 1 ) )
  then
    insert( new Fibonacci( f.sequence - 1 ) );
    System.out.println( "recurse for " + f.sequence );
end
```

この例の実行フローをさらに理解するには、**target/fibonacci.log** からの監査ログファイルを IDE デバッグビューまたは **Audit View** が利用できる場合は Audit View (例: IDE の **Window** → **Show View**) にロードします。

この例では、**監査ビュー** に、 **sequence** フィールドが **50** に指定された、**Fibonacci** の元のアサーションが表示されます。これは Java コードで実行されています。これ以降、**監査ビュー** で、ルールの再帰が継続して行われ、アサートされた **Fibonacci** オブジェクトにより、**"Recurse"** ルールがアクティベートされて、再度実行されます。

図7.6 監査ビューでのルール "Recurse"



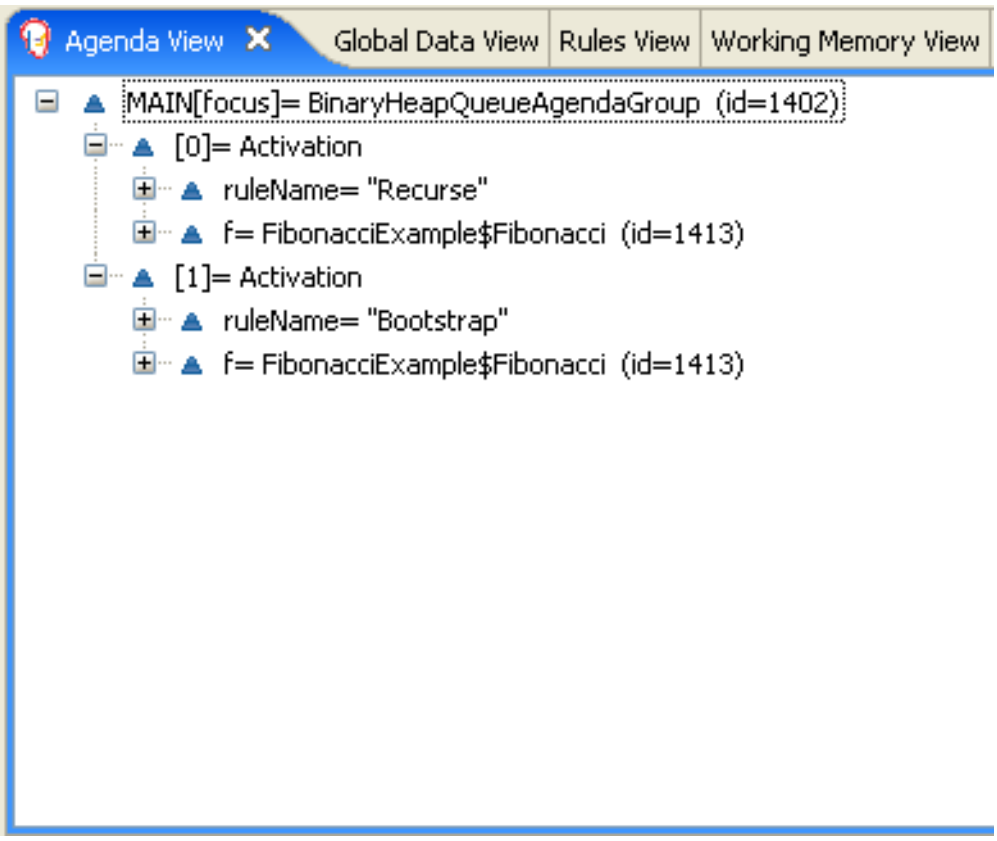
sequence フィールドが **2** の **Fibonacci** オブジェクトがアサートされると、**"Bootstrap"** ルールが一致し、**"Recurse"** ルールとともにアクティベートされます。フィールド**sequence** には、複数の制約があり、**1** または **2** と同等かをテストしている点に注目してください。

ルール "Bootstrap"

```
rule "Bootstrap"
  when
    f : Fibonacci( sequence == 1 || == 2, value == -1 ) // multi-
restriction
  then
    modify ( f ){ value = 1 };
    System.out.println( f.sequence + " == " + f.value );
end
```

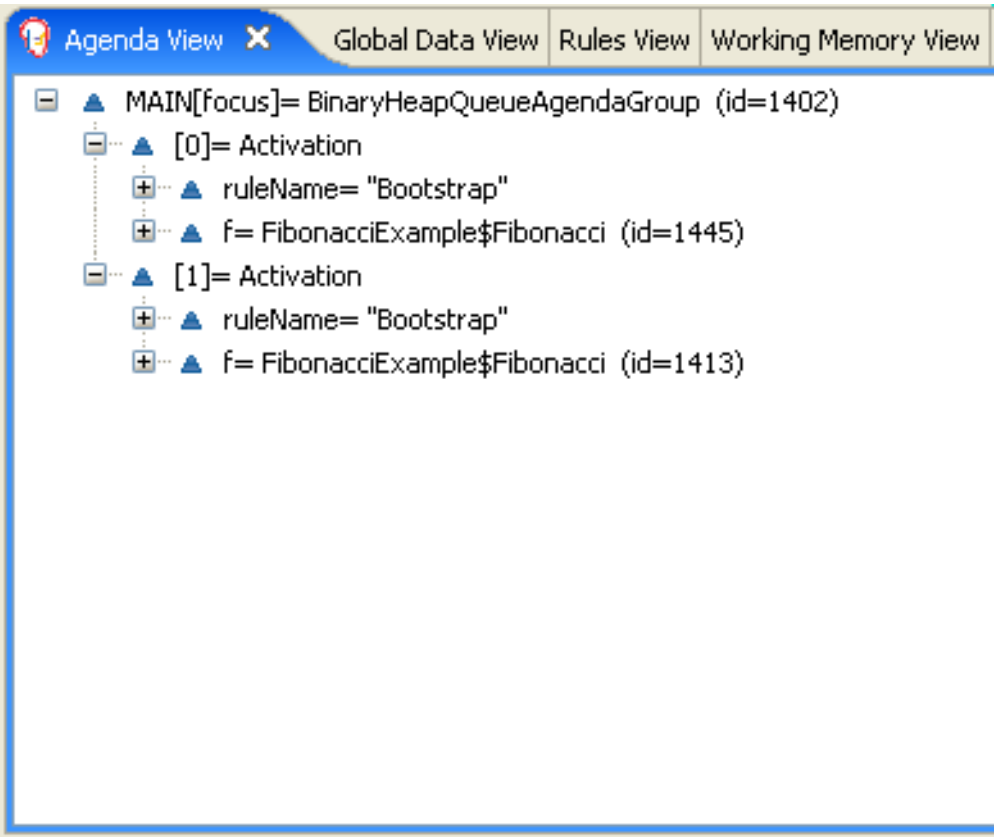
IDE で **Agenda View** を使用して、エンジンアジェンダの状態を調査できます。**"Recurse"** の顕著性の値のほうが高いので、**"Bootstrap"** ルールはまだ実行されません。

図7.7 アジェンダビュー 1 でのルール "Recurse" および "Bootstrap"



sequence が **1** の **Fibonacci** オブジェクトがアサートされると、**"Bootstrap"** ルールが再度一致し、このルールに含まれる 2 つのルールがアクティベートされます。**sequence** が **1** の **Fibonacci** オブジェクトが存在すると、すぐに **not** 条件要素で、ルールが一致しなくなるので、**"Recurse"** ルールの照合やアクティベーションはされません。

図7.8 アジェンダビュー 2 でのルール "Recurse" および "Bootstrap"



"Bootstrap" ルールは、**sequence** が **1** と **2** のオブジェクトの値を **1** に設定します。値が **-1** でない **Fibonacci** オブジェクトが 2 つあるので、**"Calculate"** ルールの照合が可能になります。

この例のある時点で、作業メモリーに 50 近くの **Fibonacci** オブジェクトが存在します。3 つ選択してそれぞれを乗算し、順番に各値を計算する必要があります。フィールドの制約なしに、ルールで Fibonacci パターン 3 つを使用してクラス積候補を絞り込む場合に、考えられる組み合わせとして 50x49x48 通りあり、約 12 万 5000 のルールを実行できるにもかかわらず、その大半が誤っていることになります。

"Calculate" ルールは、フィールドの制約を使用して正しい順番にフィボナッチパターン 3 つを評価します。この手法は **cross-product matching** と呼ばれます。

最初のパターンでは、値が **!= -1** の **Fibonacci** オブジェクトを検索して、このパターンとフィールド両方をバインドします。2 番目の **Fibonacci** オブジェクトの実行する内容は同じですが、別のフィールド制約を追加して、シーケンスが **f1** にバインドされている **Fibonacci** オブジェクトより 1 つ大きくなるようにします。このルールが初めて実行されると、シーケンスが **1** と **2** にだけ、値 **1** が割り当てられていることが分かります。また、この 2 つの制約で、**f1** がシーケンス **1** を、**f2** がシーケンス **2** を参照するようにします。

最後のパターンでは、値が **-1** と等しく、シーケンスが **f2**よりも大きい **Fibonacci** オブジェクトを検索します。

フィボナッチの例のこの時点で、3 つの **Fibonacci** オブジェクトが利用可能なクロス積から正しく選択され、**f3** にバインドされている 3 番目の **Fibonacci** オブジェクトの値を計算できます。

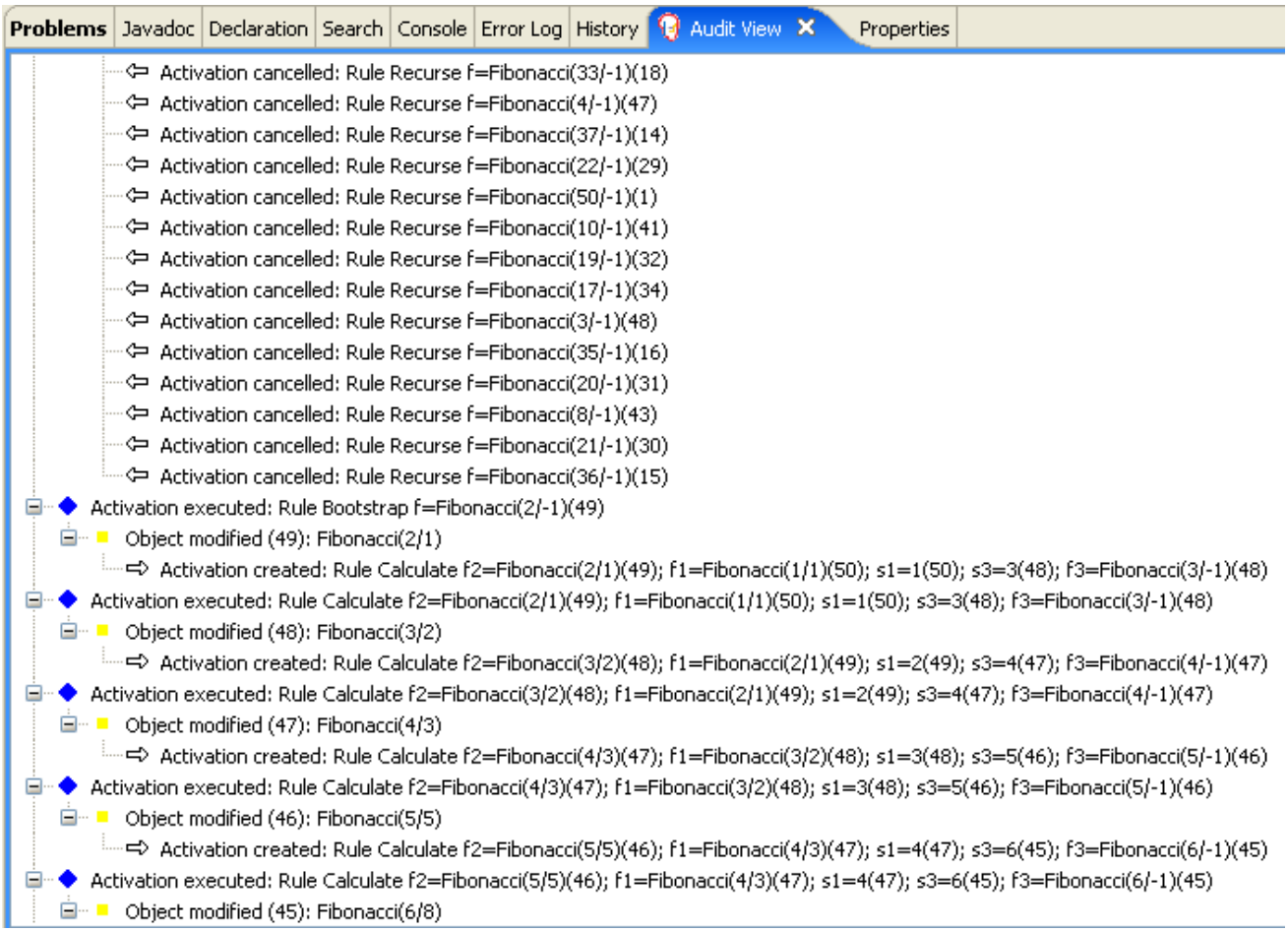
ルール "Calculate"

```
rule "Calculate"
  when
    // Bind f1 and s1.
    f1 : Fibonacci( s1 : sequence, value != -1 )
    // Bind f2 and v2, refer to bound variable s1.
    f2 : Fibonacci( sequence == (s1 + 1), v2 : value != -1 )
    // Bind f3 and s3, alternative reference of f2.sequence.
    f3 : Fibonacci( s3 : sequence == (f2.sequence + 1 ), value == -1 )
  then
    // Note the various referencing techniques.
    modify ( f3 ) { value = f1.value + v2 };
    System.out.println( s3 + " == " + f3.value );
end
```

modify ステートメントにより、**f3** にバインドされた **Fibonacci** オブジェクトの値が更新されます。つまり、値が **-1** 以外の **Fibonacci** オブジェクトが新たに存在するということで、**"Calculate"** ルールにより、再度合致があるか検索して次のフィボナッチ番号を算出することができます。

IDE のデバッグビューまたは **監査ビュー** では、最後の **"Bootstrap"** ルールが実行されることで **Fibonacci** オブジェクトが変更され、**"Calculate"** ルールに合致し、次に、別の **Fibonacci** オブジェクトが変更され、この **"Calculate"** ルールに再度合致できていることが分かります。このプロセスは、すべての **Fibonacci** オブジェクトに値が設定されるまで継続されます。

図7.9 監査ビューのルール



7.5. ペットショップの例のデシジョン (アジェンダグループ、グローバル変数、コールバック、GUI 統合)

ペットショップの例のデシジョンセットでは、ルールでのアジェンダグループとグローバル変数の使用方法と、Red Hat Process Automation Manager ルールとグラフィカルユーザーインターフェース (GUI) の統合方法が分かります。今回は Swing ベースのデスクトップアプリケーションを使用します。また、この例では、コールバックを使用して実行中のデシジョンエンジンと通信し、ランタイム時に加えられた作業メモリー内の変更をもとに GUI を更新する方法を例示しています。

以下は、ペットショップの例の概要です。

- **名前**: **`petstore`**
- **Main クラス**: (**`src/main/java`** 内の) **`org.drools.examples.petstore.PetStoreExample`**
- **モジュール**: **`drools-examples`**
- **タイプ**: Java アプリケーション
- **ルールファイル**: (**`src/main/resources`** 内の) **`org.drools.examples.petstore.PetStore.drl`**
- **目的**: ルールアジェンダグループ、グローバル変数、コールバック、GUI 統合を例示します。

ペットショップの例では、**PetStoreExample.java** クラス例を使用して (Swing イベントを処理する複数のクラスに加え)、以下のクラスを主に定義しています。

- **Petstore** には **main()** メソッドが含まれます。
- **PetStoreUI** は Swing ベースの GUI を作成して表示します。このクラスには複数の小さいクラスが含まれており、マウスボタンのクリックなど、さまざまな GUI イベントに主に対応します。
- **TableModel** には表データが含まれています。このクラスは基本的に **AbstractTableModel** を継承する JavaBean です。
- **CheckoutCallback** により、GUI がルールと対話できるようになります。
- **Ordershow** は購入するアイテムを保持します。
- **Purchase** には、顧客が購入する商品と商品の詳細が保存されます。
- **Product** は、販売可能な商品と価格の詳細を含む JavaBean です。

この例の Java コードはほぼ、プレーンな JavaBean か Swing ベースとなっています。Swing コンポーネントの詳細は、「Creating a GUI with JFC/Swing」 の Java チュートリアルを参照してください。

ペットショップの例でのルール実行動作
他の例のディジョンセットではファクトがすぐにアサートされて実行されるのに対し、ペットショップの例では、ユーザーの対話をもとに他のファクトが収集されるまでルールは実行されます。このルールでは、コンストラクターで作成される **PetStoreUI** オブジェクトを使用してルールを実行し、**Vector** オブジェクトの **stock** を受けれ入れて商品を収集します。次に、この例では、以前の読み込まれたルールベースを含む **CheckoutCallback** クラスのインスタンスを使用します。

ペットショップの KIE コンテナーおよびファクト実行の設定

```
// KieServices is the factory for all KIE services.
KieServices ks = KieServices.Factory.get();

// Create a KIE container on the class path.
KieContainer kc = ks.getKieClasspathContainer();

// Create the stock.
Vector<Product> stock = new Vector<Product>();
stock.add( new Product( "Gold Fish", 5 ) );
stock.add( new Product( "Fish Tank", 25 ) );
stock.add( new Product( "Fish Food", 2 ) );

// A callback is responsible for populating the working memory and for
firing all rules.
PetStoreUI ui = new PetStoreUI( stock,
                                new CheckoutCallback( kc ) );
ui.createAndShowGUI();
```

ルールを実行する Java コードは **CheckoutCallBack.checkout()** メソッドに含まれます。このメソッドは、ユーザーが UI で **チェックアウト** をクリックするとトリガーされます。

CheckoutCallBack.checkout() からのルール実行

```
public String checkout(JFrame frame, List<Product> items) {
    Order order = new Order();

    // Iterate through list and add to cart.
    for ( Product p: items ) {
        order.addItem( new Purchase( order, p ) );
    }

    // Add the JFrame to the ApplicationData to allow for user
interaction.

    // From the KIE container, a KIE session is created based on
    // its definition and configuration in the META-INF/kmodule.xml file.
    KieSession ksession = kcontainer.newKieSession("PetStoreKS");

    ksession.setGlobal( "frame", frame );
    ksession.setGlobal( "textArea", this.output );

    ksession.insert( new Product( "Gold Fish", 5 ) );
    ksession.insert( new Product( "Fish Tank", 25 ) );
    ksession.insert( new Product( "Fish Food", 2 ) );

    ksession.insert( new Product( "Fish Food Sample", 0 ) );

    ksession.insert( order );

    // Execute rules.
    ksession.fireAllRules();

    // Return the state of the cart
    return order.toString();
}
```

このコード例では、2 つの要素を **CheckoutCallBack.checkout()** メソッドに渡します。1 つ目の要素は、GUI の一番下にある出力テキストのフレームを囲む **JFrame** Swing コンポーネントのハンドルです。2 つ目の要素は注文アイテムのリストで、GUI の右上のセクションにある **Table** エリアからの情報を保存する **TableModel** から取得します。

for ループは GUI からの注文アイテム一覧を **Order** JavaBean に変換します。これは、**PetStoreExample.java** ファイルにも含まれています。

今回の例では、データはすべて Swing コンポーネントに含まれており、ユーザーが UI の **チェックアウト** をクリックしない限り実行されないため、ルールはステートレスの KIE セッションで実行します。ユーザーが **チェックアウト** をクリックするたびに、リストの内容を Swing **TableModel** から KIE セッションの作業メモリーに移動し、**ksession.fireAllRules()** メソッドで実行します。

このコード内には、**KieSession** への呼び出しが 9 個あります。1 つ目は、**KieContainer** から新しい **KieSession** を作成します (この例では、**main()** メソッドの **CheckoutCallBack** クラスから **KieContainer** に渡されます)。次の 2 つの呼び出しは、ルールでグローバル変数として保持されるオブジェクト を 2 つ渡します (メッセージの書き込みに使用する Swing テキストエリアと Swing フレーム)。他に挿入することで、商品の情報を **KieSession** と注文リストに配置します。最後の呼び出しは、標準の **fireAllRules()** です。

ペットショップのルールファイルのインポート、グローバル変数、Java 関数

PetStore.drl ファイルには、さまざまな Java クラスをルールで利用できるように、標準のパッケー

ジとインポートステートメントが含まれています。このルールファイルには、**frame** および **textArea** などのように、ルール内で使用する **グローバル変数** が含まれています。グローバル変数では、Swing コンポーネント **JFrame** と、**setGlobal()** メソッドを呼び出した Java コードにより以前に渡された **JTextArea** コンポーネントへの参照を保持します。ルールが実行されるとすぐに失効するルールの標準変数とは異なり、グローバル変数は KIE セッションの有効期間中この値を保持します。つまり、この後に続く全ルールを評価するのに、これらの変数の内容を使用できます。

PetStore.drl パッケージ、インポートおよびグローバル変数

```
package org.drools.examples;

import org.kie.api.runtime.KieRuntime;
import org.drools.examples.petstore.PetStoreExample.Order;
import org.drools.examples.petstore.PetStoreExample.Purchase;
import org.drools.examples.petstore.PetStoreExample.Product;
import java.util.ArrayList;
import javax.swing.JOptionPane;

import javax.swing.JFrame;

global JFrame frame
global javax.swing.JTextArea textArea
```

PetStore.drl ファイルには、このファイル内のルールが使用する関数 2 つも含まれています。

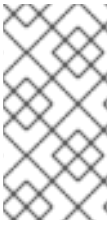
PetStore.drl Java 関数

```
function void doCheckout(JFrame frame, KieRuntime krt) {
        Object[] options = {"Yes",
                            "No"};

        int n = JOptionPane.showOptionDialog(frame,
                                             "Would you like to
checkout?",
                                             "",
                                             JOptionPane.YES_NO_OPTION,

JOptionPane.QUESTION_MESSAGE,
                                             null,
                                             options,
                                             options[0]);

       if (n == 0) {
            krt.getAgenda().getAgendaGroup( "checkout" ).setFocus();
       }
}

function boolean requireTank(JFrame frame, KieRuntime krt, Order order,
Product fishTank, int total) {
        Object[] options = {"Yes",
                            "No"};

        int n = JOptionPane.showOptionDialog(frame,
                                             "Would you like to buy a
tank for your " + total + " fish?",
```

```
                                                "Purchase Suggestion",
                                                JOptionPane.YES_NO_OPTION,

JOptionPane.QUESTION_MESSAGE,
                                                null,
                                                options,
                                                options[0]);

        System.out.print( "SUGGESTION: Would you like to buy a tank for
your "
                                + total + " fish? - " );

        if (n == 0) {
              Purchase purchase = new Purchase( order, fishTank );
              krt.insert( purchase );
              order.addItem( purchase );
              System.out.println( "Yes" );
        } else {
             System.out.println( "No" );
        }
        return true;
}
```

この 2 つの関数は以下のアクションを実行します。

- **doCheckout()** は、チェックアウトするかどうかユーザーに尋ねるダイアログボックスを表示します。チェックアウトする場合は、フォーカスが **checkout** アジェンダグループに設定され、そのグループのルールが (今後) 実行できるようにします。
- **requireTank()** は、水槽を購入するかどうかを確認するダイアリーを表示します。購入する場合は、新しい水槽の **Product** が作業メモリーの注文リストに追加されます。



注記

この例では、効率化を図るため、すべてのルールと関数が同じルールファイルで実行しています。実稼働環境では、通常、ルールと関数を別のファイルに分けるか、静的な Java メソッドを構築して、**import function my.package.name.hello** などのインポート関数を使用し、ファイルをインポートします。

アジェンダグループを使用したペットショップルール

ペットショップの例のルールはほぼ、アジェンダグループを使用してルールの実行を制御しています。アジェンダグループを使用すると、エンジンアジェンダを分割し、ルールのグループの実行を、詳細にわたり制御できるようになります。デフォルトでは、全ルールはアジェンダグループ **MAIN** に含まれます。**agenda-group** 属性を使用してルールに異なるアジェンダグループを指定できます。

最初は、作業メモリーは、アジェンダグループ **MAIN** にフォーカスを当てます。アジェンダグループのルールは、グループがこのフォーカスを受けた場合のみ実行されます。**setFocus()** のメソッドか、**auto-focus** のルール属性を使用してフォーカスを設定できます。**auto-focus** 属性を使用すると、ルールが一致してアクティベートされた場合のみ、ルールにアジェンダグループのフォーカスが自動的に当てられます。

ペットショップの例では、ルールに以下のアジェンダグループを使用します。

- **"init"**

- **"evaluate"**
- **"show items"**
- **"checkout"**

たとえば、同じルール **"Explode Cart"** は **"init"** のアジェンダグループを使用して、ショッピングカードのアイテムを実行して、KIE セッションの作業メモリーに挿入するオプションが提供されるようにします。

ルール "Explode Cart"

```
// Insert each item in the shopping cart into the working memory.
rule "Explode Cart"
    agenda-group "init"
    auto-focus true
    salience 10
    dialect "java"
  when
    $order : Order( grossTotal == -1 )
    $item : Purchase() from $order.items
  then
    insert( $item );
    kcontext.getKnowledgeRuntime().getAgenda().getAgendaGroup( "show
items" ).setFocus();
    kcontext.getKnowledgeRuntime().getAgenda().getAgendaGroup( "evaluate"
).setFocus();
end
```

このルールは、**grossTotal** がまだ計算されていない全注文に対して照合が行われます。購入アイテムごとに、順番に実行がループされます。

ルールは、アジェンダグループに関連する以下の機能を使用します。

- **agenda-group "init"** はアジェンダグループの名前を定義します。この例では、グループにはルールが 1 つしかありませんが、Java コードもルール結果もこのグループにフォーカスされていないため、**auto-focus** の属性により、ルールが実行されるかが決まります。
- このルールはアジェンダグループで唯一のルールですが、**auto-focus true** を使用して、**fireAllRules()** が Java コードから呼び出されると、必ず実行されるようにします。
- **kcontext….setFocus()** で、フォーカスを **"show items"** と **"evaluate"** のアジェンダグループに設定して、これらのルールが実行されるようにします。実際は、注文に含まれる全アイテムをループでチェックし、メモリーに挿入してから、挿入ごとに他のルールを実行します。

"show items" アジェンダグループには **"Show Items"** というルール 1 つだけが含まれます。KIE セッションの作業メモリーに現在含まれる注文で購入があるたびに、このルールを使用して、ルールファイルに定義した **textArea** 変数をもとに、GUI の下の部分にあるテキストエリアに詳細がロギングされます。

ルール "Show Items"

```
rule "Show Items"
    agenda-group "show items"
```

```
    dialect "mvel"
  when
    $order : Order( )
    $p : Purchase( order == $order )
  then
   textArea.append( $p.product + "\n");
end
```

また、**"evaluate"** アジェンダグループにより、**"Explode Cart"** ルールからフォーカスを取得します。このアジェンダグループには、**"Free Fish Food Sample"** と **"Suggest Tank"** のルールが 2 つ含まれます。"Free Fish Food Sample"、"Suggest Tank" の順番に実行されます。

ルール "Free Fish Food Sample"

```
// Free fish food sample when users buy a goldfish if they did not already
buy
// fish food and do not already have a fish food sample.
rule "Free Fish Food Sample"
    agenda-group "evaluate" (1)
    dialect "mvel"
  when
    $order : Order()
    not ( $p : Product( name == "Fish Food") && Purchase( product == $p )
) (2)
    not ( $p : Product( name == "Fish Food Sample") && Purchase( product
== $p ) ) (3)
    exists ( $p : Product( name == "Gold Fish") && Purchase( product == $p
) ) (4)
    $fishFoodSample : Product( name == "Fish Food Sample" );
  then
    System.out.println( "Adding free Fish Food Sample to cart" );
    purchase = new Purchase($order, $fishFoodSample);
    insert( purchase );
    $order.addItem( purchase );
end
```

ルール **"Free Fish Food Sample"** は、以下の条件がすべて該当する場合のみ実行されます。

(1) アジェンダグループ **"evaluate"** がルール実行で評価さている

(2) ユーザーが魚の餌をまだ持っていない

(3) ユーザーが無料の魚の餌サンプルをまだ持っていない

(4) ユーザーが金魚を注文している

この注文ファクトが上記の要件すべてを満たす場合には、新しい商品 (Fish Food Sample) が作成され、作業メモリーの注文に追加されます。

ルール "Suggest Tank"

```
// Suggest a fish tank if users buy more than five goldfish and
// do not already have a tank.
rule "Suggest Tank"
```

```
    agenda-group "evaluate"
    dialect "java"
  when
    $order : Order()
    not ( $p : Product( name == "Fish Tank") && Purchase( product == $p )
) (1)
    ArrayList( $total : size > 5 ) from collect( Purchase( product.name ==
"Gold Fish" ) ) (2)
    $fishTank : Product( name == "Fish Tank" )
  then
    requireTank(frame, kcontext.getKieRuntime(), $order, $fishTank,
$total);
end
```

ルール **"Suggest Tank"** は以下の条件がすべて該当する場合のみ実行されます。

1. ユーザーが水槽を注文していない
2. ユーザーが 6 匹以上注文した

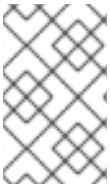
このルールが実行されると、ルールファイルに定義されている **requireTank()** 関数が呼び出されます。この関数により、水槽を購入するかどうかを尋ねるダイアログが表示されます。新しい水槽の **Product** が作業メモリーの注文リストに追加されます。ルールが **requireTank()** 関数を呼び出した場合には、このルールを使用して、関数に Swing GUI のハンドルが含まれるように、**frame** のグローバル変数を渡します。

ペットショップの例の **"do checkout"** ルールにはアジェンダルールや **when** 条件がないので、ルールは常に実行されて、デフォルトの **MAIN** のアジェンダグループの一部とみなされます。

ルール "do checkout"

```
rule "do checkout"
    dialect "java"
  when
  then
    doCheckout(frame, kcontext.getKieRuntime());
end
```

このルールが実行されると、ルールファイルで定義されている **doCheckout()** 関数を呼び出します。この関数により、チェックアウトするかどうかユーザーに尋ねるダイアログボックスが表示されます。チェックアウトする場合は、フォーカスが **checkout** アジェンダグループに設定され、そのグループのルールが (今後) 実行できるようにします。このルールで **doCheckout()** 関数を呼び出し、この変数に Swing GUI のハンドルが含まれるように **frame** グローバル変数を渡します。



注記

この例では、結果が想定どおりに実行されない場合のトラブルシューティングの方法を例示します。ルールの **when** ステートメントから条件を削除して、**then** ステートメントのアクションをテストし、アクションが正しく実行されることを検証します。

"checkout" アジェンダグループには、**"Gross Total"**、**"Apply 5% Discount"** および **"Apply 10% Discount"** の注文のチェックアウト処理、割引の適用のルールが 3 つ含まれています。

ルール "Gross Total"、"Apply 5% Discount" および "Apply 10% Discount"

```
rule "Gross Total"
    agenda-group "checkout"
    dialect "mvel"
  when
    $order : Order( grossTotal == -1)
    Number( total : doubleValue ) from accumulate( Purchase( $price :
product.price ),
                                                   sum(
$price ) )
  then
    modify( $order ) { grossTotal = total }
    textArea.append( "\ngross total=" + total + "\n" );
end

rule "Apply 5% Discount"
    agenda-group "checkout"
    dialect "mvel"
  when
    $order : Order( grossTotal >= 10 && < 20 )
  then
    $order.discountedTotal = $order.grossTotal * 0.95;
    textArea.append( "discountedTotal total=" + $order.discountedTotal +
"\n" );
end

rule "Apply 10% Discount"
    agenda-group "checkout"
    dialect "mvel"
  when
    $order : Order( grossTotal >= 20 )
  then
    $order.discountedTotal = $order.grossTotal * 0.90;
    textArea.append( "discountedTotal total=" + $order.discountedTotal +
"\n" );
end
```

ユーザーがまだ総計を算出していない場合には、**`Gross Total`** で、商品の価格を累積して合計を出し、この合計を KIE セッションに渡して、**`textArea`** のグローバル変数を使用し、Swing **`JTextArea`** で合計を表示します。

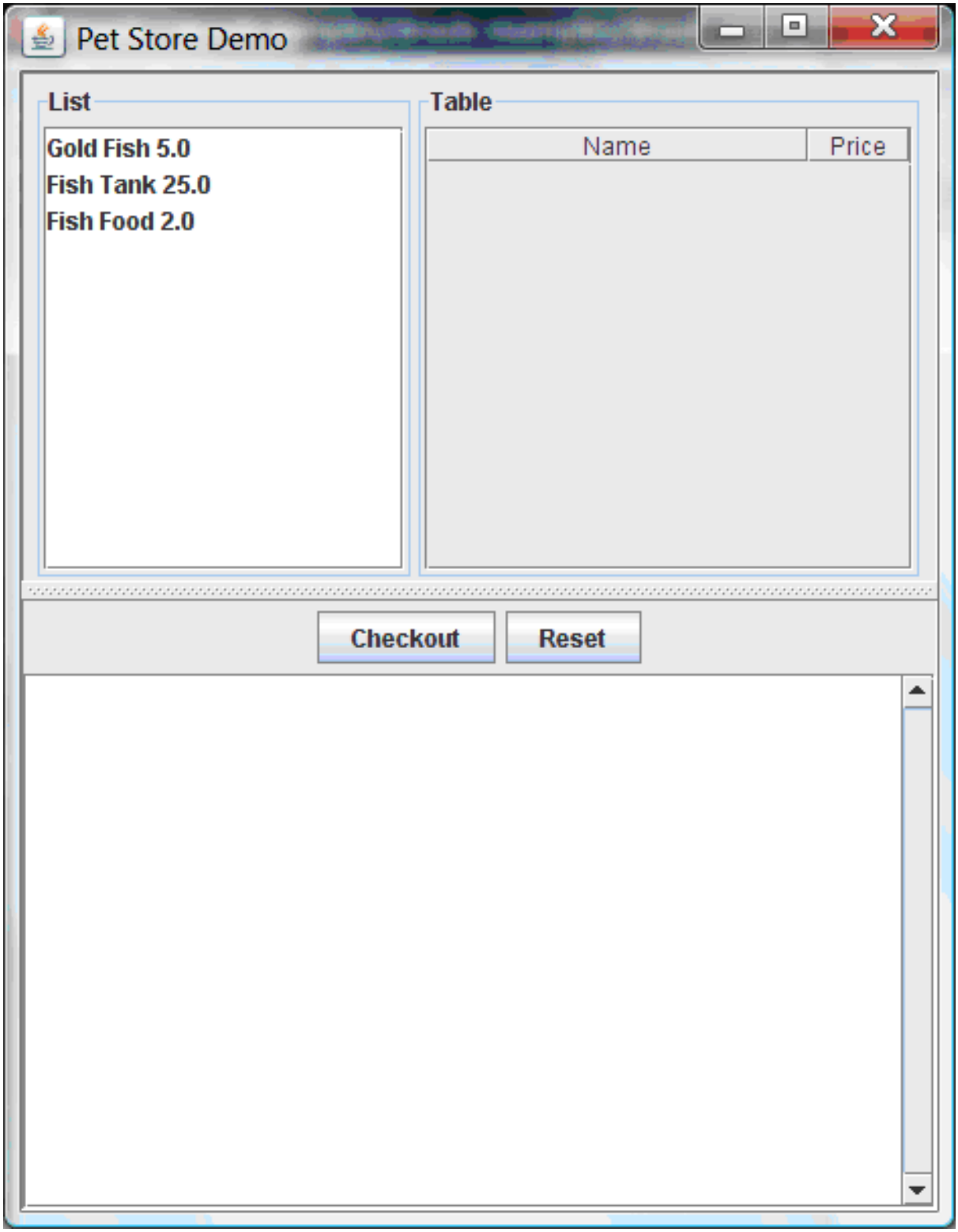
総計が **`10`** から **`20`** (通貨単位) の場合には、**`"Apply 5% Discount"`** ルールで割引合計を計算し、KIE セッションに追加して、テキストエリアに表示します。

総計が **`20`** 未満の場合には、**`"Apply 10% Discount"`** ルールで割引合計を計算し、KIE セッションに追加して、テキストエリアに表示します。

ペットショップ例の実行
他の Red Hat Process Automation Manager のデシジョン例と同じように、お使いの IDE で **`org.drools.examples.petstore.PetStoreExample`** クラスを Java アプリケーションとして実行し、ペットショップの例を実行します。

ペットショップの例を実行すると、**Pet Store Demo** GUI ウィンドウが表示されます。このウィンドウでは、購入可能な商品 (左上)、選択済み商品の空白のリスト (右上)、**チェックアウト** および **リセット** ボタン (真ん中)、空白のシステムメッセージエリア (下) が表示されます。

図7.10 起動後のペットショップ例の GUI



この例では、以下のイベントが発生して、この実行動作を確立します。

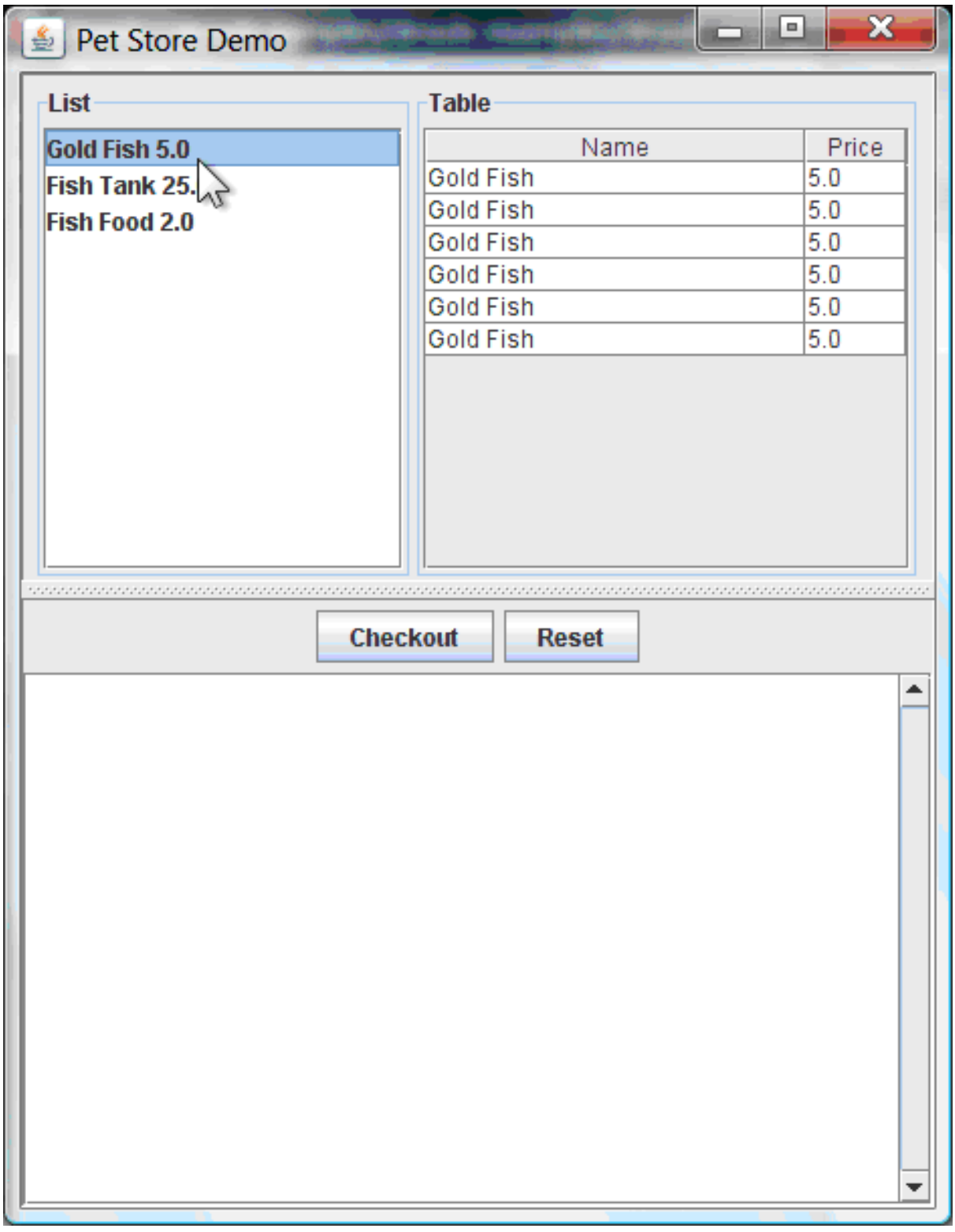
1. **main()** メソッドがルールベースの実行と読み込みを終えているが、ルールが実行されていないこと。今のところ、実行されたルールに関連する唯一のコードがこれです。

2. 新しい **`PetStoreUI`** オブジェクトが作成され、後で使用できるようにルールベースにハンドルを渡すこと。

3. さまざまな Swing コンポーネントが関数を実行し、最初の UI 画面が表示され、ユーザーの入力を待ちます。

リストからさまざまな商品をクリックして、UI 設定をチェックできます。

図7.11 ペットショップ例の GUI のチェック



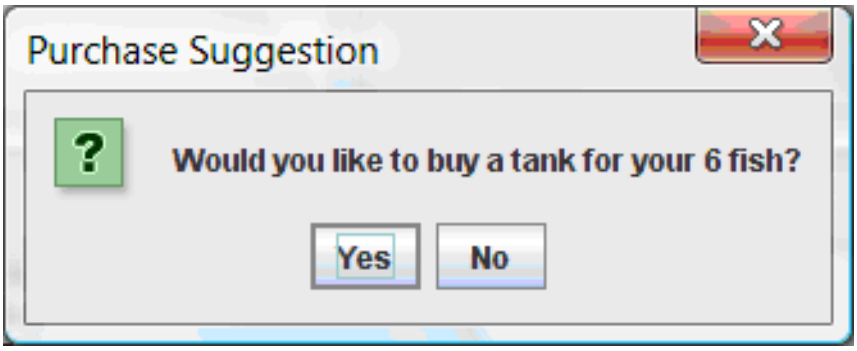
ルールコードはまだ実行されていません。UI は Swing コードを使用してユーザーによるマウスクリックを検出し、選択済みの商品を **TableModel** オブジェクトに追加して、UI の右上隅に表示します。この例では、Model-View-Controller 設計パターンを紹介しています。

チェックアウト をクリックすると、ルールが以下の方法で実行されます。

1. Swing クラスは **チェックアウト** がクリックされるまで待機して、(最終的に) **CheckOutCallBack.checkout()** メソッドを呼び出します。これにより、**TableModel** オブジェクト (UI の右上隅) から KIE セッションの作業メモリーにデータを挿入します。その後に、メソッドによりルールが実行されます。

2. **"Explode Cart"** ルールは、**auto-focus** 属性を **true** に設定して最初に実行します。このルールは、カートの商品すべてを順にループしていき、商品が作業メモリーに含まれていることを確認し、**"show Items"** と **"evaluate"** アジェンダグループに実行するオプションを提供します。このグループのルールは、カートのコンテンツをテキストエリア (UI の下) に追加して、魚の餌を無料で受け取る資格があるかどうかを評価し、また水槽購入の有無を尋ねるかどうかを決定します。

   図7.12 水槽の資格



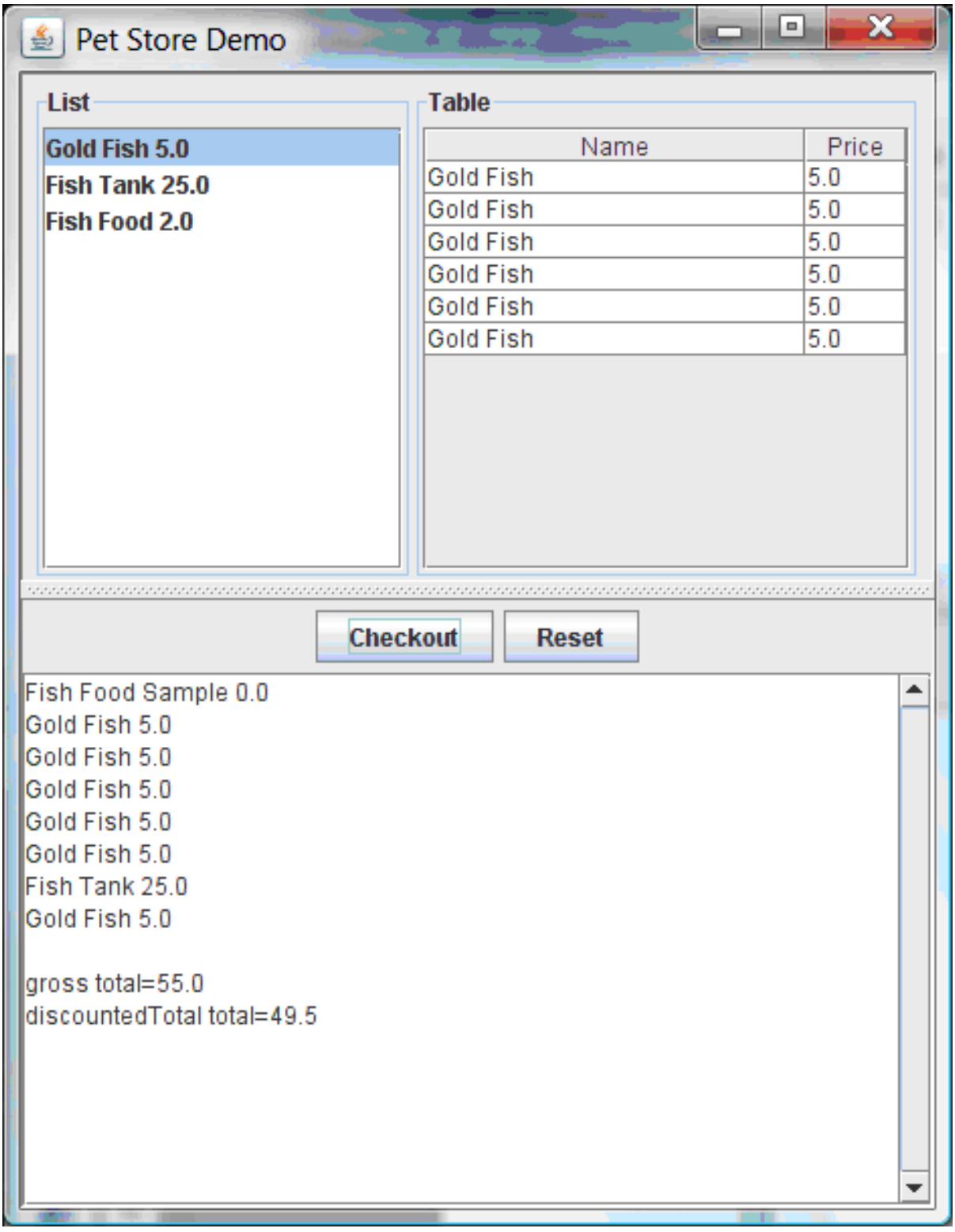
3. 現在、他のアジェンダグループにフォーカスが当たっておらず、**"do checkout"** ルールは、デフォルトの **MAIN** アジェンダグループに含まれているので、次に実行されます。このルールは常に **doCheckout()** 関数を呼び出し、この関数によりチェックアウトをするかどうかが尋ねられます。

4. **doCheckout()** 関数は、フォーカスを **"checkout"** アジェンダグループに設定し、そのグループ内のルールに、実行するオプションを提供します。

5. **"checkout"** アジェンダグループ内のルールは、「カート」内の内容を表示し、適切な割引を適用します。

6. Swing は、別の商品の選択 (およびもう一度ルールを実行させる) または GUI の終了のいずれかのユーザー入力を待ちます。

図7.13 全ルールが実行された後のペットショップ例の GUI



IDE コンソールでイベントのこのフローを例示するには、他の **System.out** 呼び出しを追加します。

IDE コンソールの System.out 出力

```
Adding free Fish Food Sample to cart
SUGGESTION: Would you like to buy a tank for your 6 fish? - Yes
```

7.6. 誠実な政治家の例のデシジョン (真理維持および顕著性)

誠実な政治家の例におけるデシジョンセットでは、論理挿入を使用した真理維持の概念およびルールでの顕著性の使用方法を紹介します。

以下は、誠実な政治家の例の概要です。

- **名前**: **honestpolitician**
- **Main クラス**: (**src/main/java** 内の) **org.drools.examples.honestpolitician.HonestPoliticianExample**
- **モジュール**: **drools-examples**
- **タイプ**: Java アプリケーション
- **ルールファイル**: (**src/main/resources** 内の) **org.drools.examples.honestpolitician.HonestPolitician.drl**
- **目的**: ファクトの論理挿入をもとにした真理維持の概念およびルールでの顕著性の使用方法を紹介します。

誠実な政治家例の前提として基本的に、ステートメントが True の場合にのみ、オブジェクトが存在できます。**insertLogical()** メソッドを使用して、ルールの結果により、オブジェクトを論理的に挿入します。つまり、論理的に挿入されたルールが True の状態であれば、オブジェクトは KIE セッションの作業メモリー内に留まります。ルールが True でなくなると、オブジェクトは自動的に取り消されます。

この例では、ルールを実行することで、企業による政治家の買収が原因で、政治家グループが「誠実」から「不誠実」に変わります。各政治家が評価されるにつれ、最初は honesty 属性を **true** に設定して開始しますが、ルールが実行されると政治家は「誠実」ではなくなります。状態が「誠実」から「不誠実」に切り替わると、作業メモリーから削除されます。ルールの顕著性により、顕著性が定義されているルールをどのように優先付けするかを、エンジンに通知します。そうでない場合には、デフォルトの顕著性の値 **0** を使用します。アクティベーションキューで順番待ちをしている場合には、顕著性の高いルールの優先順位が高くなります。

Politician および Hope クラス

この例の **Politician** クラス例は、誠実な政治家として設定されています。**Politician** クラスは、文字列アイテム **name** とブール値アイテム **honest** で構成されています。

Politician クラス

```
public class Politician {
    private String name;
    private boolean honest;
    ...
}
```

Hope クラスは、**Hope** オブジェクトが存在するかどうかを判断します。このクラスには意味を持つメンバーは存在しませんが、社会に希望がある限り、作業メモリーに存在します。

Hope クラス

```
public class Hope {

    public Hope() {
```

```
    }
  }
```

政治家の誠実性に関するルール定義
誠実な政治家の例では、作業メモリーに最低でも 1 名誠実な政治家が存在する場合には、**"We have an honest Politician"** ルールで論理的に新しい **Hope** オブジェクトを挿入します。すべての政治家が不誠実になると、**Hope** オブジェクトは自動的に取り除かれます。このルールでは、**salience** 属性の値が **10** となっており、他のルールより先に実行されます。理由は、この時点では **"Hope is Dead"** ルールが True となっているためです。

ルール "We have an honest politician"

```
rule "We have an honest Politician"
    salience 10
  when
    exists( Politician( honest == true ) )
  then
    insertLogical( new Hope() );
end
```

Hope オブジェクトが存在すると、すぐに **"Hope Lives"** ルールが一致して実行されます。**"Corrupt the Honest"** ルールよりも優先されるように、このルールにも **salience** 値を **10** に指定しています。

ルール "Hope Lives"

```
rule "Hope Lives"
    salience 10
  when
    exists( Hope() )
  then
    System.out.println("Hurrah!!! Democracy Lives");
end
```

最初は、誠実な政治家が 4 人いるので、このルールには 4 つのアクティベーションが存在し、すべてが競合しています。各ルールが順番に実行され、政治家が誠実でなくなるように、企業により各政治家を買収させていきます。政治家 4 人すべてが買収されたら、プロパティーが **honest == true** の政治家はいなくなります。**"We have an honest Politician"** のルールは True でなくなり、論理的に挿入されるオブジェクト (最後に実行された **new Hope()** による) は自動的に取り除かれます。

ルール "Corrupt the Honest"

```
rule "Corrupt the Honest"
  when
    politician : Politician( honest == true )
    exists( Hope() )
  then
    System.out.println( "I'm an evil corporation and I have corrupted " +
politician.getName() );
    modify ( politician ) { honest = false };
end
```

真理維持システムにより **Hope** オブジェクトが自動的に取り除かれると、**Hope** に適用された条件付き要素 **not** は True でなくなり、**"Hope is Dead"** ルールが一致して実行されます。

ルール "Hope is Dead"

```
rule "Hope is Dead"
  when
    not( Hope() )
  then
    System.out.println( "We are all Doomed!!! Democracy is Dead" );
end
```

実行と監査証跡
HonestPoliticianExample.java クラスでは、honest の状態が **true** に設定されている政治家 4 人が挿入され、定義したビジネスルールに対して評価します。

HonestPoliticianExample.java クラスの実行

```
public static void execute( KieContainer kc ) {
        KieSession ksession = kc.newKieSession("HonestPoliticianKS");

        final Politician p1 = new Politician( "President of Umpa Lumpa",
true );
        final Politician p2 = new Politician( "Prime Minster of
Cheeseland", true );
        final Politician p3 = new Politician( "Tsar of Pringapopaloo",
true );
        final Politician p4 = new Politician( "Omnipotence Om", true );

        ksession.insert( p1 );
        ksession.insert( p2 );
        ksession.insert( p3 );
        ksession.insert( p4 );

        ksession.fireAllRules();

        ksession.dispose();
    }
```

この例を実行するには、IDE で Java アプリケーションとして **org.drools.examples.honestpolitician.HonestPoliticianExample** クラスを実行します。

実行後に、以下の出力が IDE コンソールウィンドウに表示されます。

IDE コンソールでの実行出力

```
Hurrah!!! Democracy Lives
I'm an evil corporation and I have corrupted President of Umpa Lumpa
I'm an evil corporation and I have corrupted Prime Minster of Cheeseland
I'm an evil corporation and I have corrupted Tsar of Pringapopaloo
I'm an evil corporation and I have corrupted Omnipotence Om
We are all Doomed!!! Democracy is Dead
```

この出力では、democracy lives に誠実な政治家が最低でも 1 人いることが分かります。ただし、各政治家は企業に買収されているので、全政治家は不誠実になり、民主性がなくなります。

この例の実行フローをさらに理解するには、**HonestPoliticianExample.java** クラスを変更して、**RuleRuntime** リスナーと監査ロガーを追加して、実行の詳細を表示できます。

監査ロガーを含む HonestPoliticianExample.java クラス

```
package org.drools.examples.honestpolitician;

import org.kie.api.KieServices;
import org.kie.api.event.rule.DebugAgendaEventListener; (1)
import org.kie.api.event.rule.DebugRuleRuntimeEventListener;
import org.kie.api.runtime.KieContainer;
import org.kie.api.runtime.KieSession;

public class HonestPoliticianExample {

    /**
     * @param args
     */
    public static void main(final String[] args) {
     KieServices ks = KieServices.Factory.get(); (2)
     //ks = KieServices.Factory.get();
        KieContainer kc =
KieServices.Factory.get().getKieClasspathContainer();
        System.out.println(kc.verify().getMessages().toString());
        //execute( kc );
        execute( ks, kc); (3)
    }

    public static void execute( KieServices ks, KieContainer kc ) { (4)
        KieSession ksession = kc.newKieSession("HonestPoliticianKS");

        final Politician p1 = new Politician( "President of Umpa Lumpa",
true );
        final Politician p2 = new Politician( "Prime Minster of
Cheeseland", true );
        final Politician p3 = new Politician( "Tsar of Pringapopaloo",
true );
        final Politician p4 = new Politician( "Omnipotence Om", true );

        ksession.insert( p1 );
        ksession.insert( p2 );
        ksession.insert( p3 );
        ksession.insert( p4 );

        // The application can also setup listeners (5)
        ksession.addEventListener( new DebugAgendaEventListener() );
        ksession.addEventListener( new DebugRuleRuntimeEventListener() );

        // Set up a file-based audit logger.
        ks.getLoggers().newFileLogger( ksession,
"./target/honestpolitician" ); (6)
```

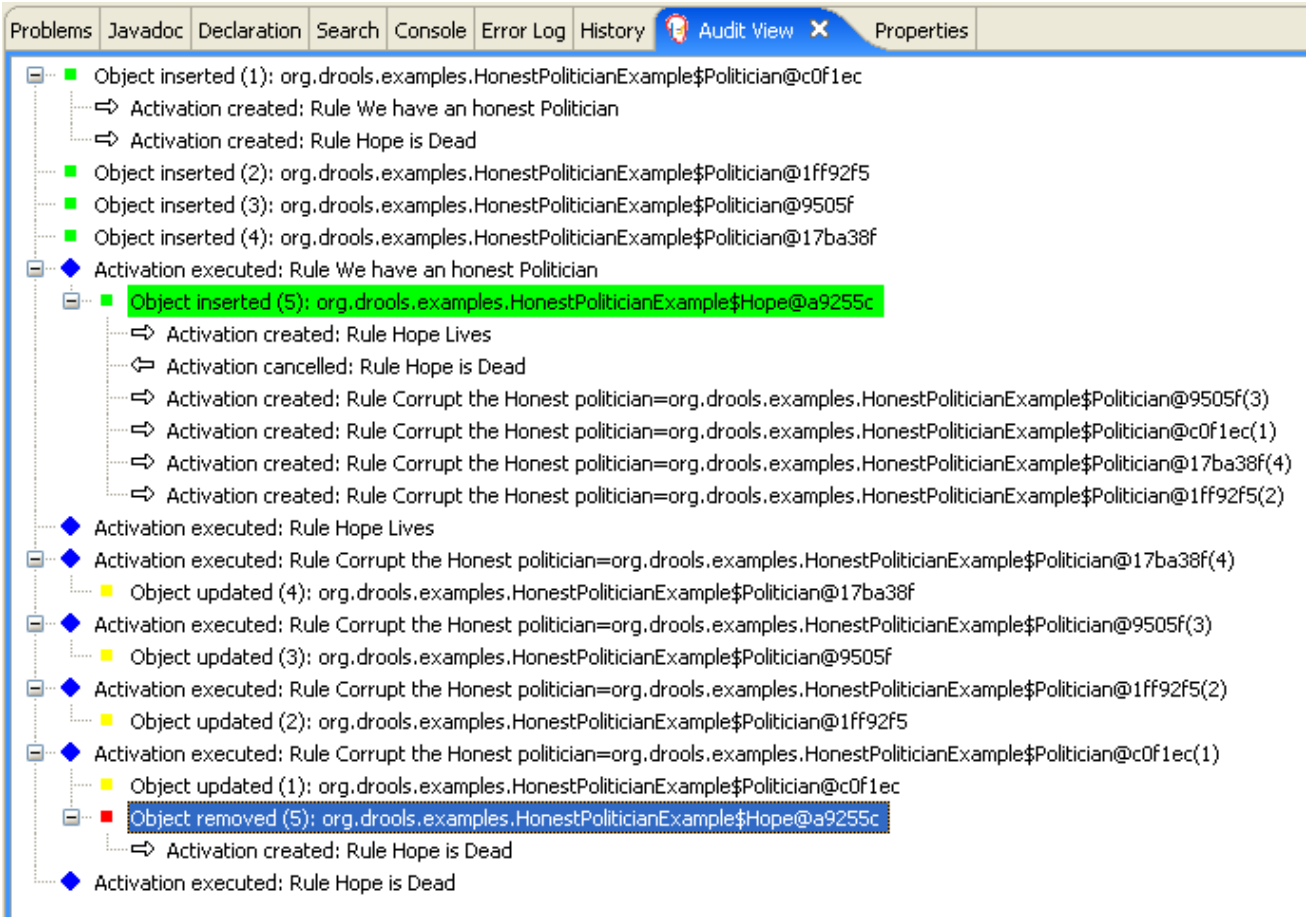
```
            ksession.fireAllRules();

            ksession.dispose();
        }

    }
```

1. **DebugAgendaEventListener** と **DebugRuleRuntimeEventListener** を処理するインポートパッケージに追加します。
2. この監査ログは **KieContainer** レベルでは利用できないので、**KieServices Factory** および **ks** 要素を作成してログを生成します。
3. **execute** メソッドを変更して **KieServices** と **KieContainer** 両方を使用します。
4. **execute** メソッドを変更して **KieContainer** に加えて **KieServices** で渡します。
5. リスナーを作成します。
6. ルールの実行後にデバッグビュー、**監査ビュー** または IDE に渡すことが可能なログを構築します。

ロギング機能を変更して、「誠実な政治家」のサンプルを実行すると、**target/honestpolitician.log** から IDE デバッグビューまたは 利用可能な場合には (IDE の一部では **Window** → **Show View**) **監査ビュー** に、監査ログファイルを読み込むことができます。

この例では、**監査ビュー** では、クラスやルールのサンプルで定義されているように、実行フロー、挿入、取り消しが示されています。

図7.14 誠実な政治家例の監査ビュー



最初の政治家が挿入されると、2 つのアクティベーションが発生します。**"We have an honest Politician"** のルールは、**exists** の条件付き要素を使用するので、最初に挿入された政治家に対してのみ一度だけアクティベートされます。この条件付き要素は、政治家が最低でも 1 人挿入されると一致します。**Hope** オブジェクトがまだ挿入されていないので、ルール **"Hope is Dead"** もこの時点でアクティベートされます。**"We have an honest Politician"** ルールは、**"Hope is Dead"** ルールより、**salience** の値が高いので先に実行され、**Hope** オブジェクト (緑にハイライト) を挿入します。**Hope** オブジェクトを挿入すると、ルール **"Hope Lives"** がアクティベートされ、ルール **"Hope is Dead"** が無効になります。この挿入により、挿入された誠実な各政治家に対して **"Corrupt the Honest"** ルールがアクティベートされます。**"Hope Lives"** のルールが実行されて、**"Hurrah!!! Democracy Lives"** が出力されます。

次に、政治家ごとに **"Corrupt the Honest"** ルールを実行して **"I'm an evil corporation and I have corrupted X"** と出力します。**X** には政治家の名前が入り、政治家の誠実性の値を **false** に変更します。最後の政治家が買収された時点で、真理維持システム (青でハイライト) により **Hope** が自動的に取り消されます。緑でハイライトされたエリアは、現在選択されている青のハイライトエリアの出元です。**Hope** ファクトが取り消されると、**"Hope is dead"** ルールが実行されて **"We are all Doomed!!! Democracy is Dead"** が出力されます。

7.7. 数独例のデシジョン (複雑なパターン一致、コールバック、GUI 統合)

数独例のディジョンセットは、人気の数字パズルゲーム「数独」をもとにしています。このセットでは、Red Hat Process Automation Manager のルールを使用してさまざまな制約をもとに、多数の考えられる回答スペースの中で回答を導き出す方法を例示します。またこの例では、Red Hat Process Automation Manager ルールとグラフィカルユーザーインターフェース (GUI) の統合方法が分かりま

す。今回は Swing ベースのデスクトップアプリケーションを使用します。また、この例では、コールバックを使用して実行中のデシジョンエンジンと通信し、ランタイム時に加えられた作業メモリー内の変更をもとに GUI を更新する方法を例示しています。

以下は数独の例の概要です。

- **名前**: **sudoku**
- **Main クラス**: (**src/main/java** 内の) **org.drools.examples.sudoku.SudokuExample**
- **モジュール**: **drools-examples**
- **タイプ**: Java アプリケーション
- **ルールファイル**: (**src/main/resources** 内の) **org.drools.examples.sudoku.*.drl**
- **目的**: 複雑なパターン一致、問題解決、コールバック、GUI 統合を例示します。

数独は、ロジックベースの数字配置パズルです。目的は、各列、行、および 3x3 ゾーン に 1 から 9 の数字が一度だけ含まれるように 9x9 のグリッドを埋めることです。パズルセッターでは、グリッド内の一部だけ記入されており、上記の制約ですべての空白を埋めるのがパズルの回答者のタスクです。

問題解決の一般的なストラテジーとして、新しい番号の挿入時に、特定の 3x3 ゾーン、行、列で同じ番号がないことを確認します。この数独例のデシジョンセットでは、Red Hat Process Automation Manager ルールを使用して、さまざまな難易度の数独パズルを解き、無効なエントリーが含まれ、不備のあるパズルの解決を試みます。

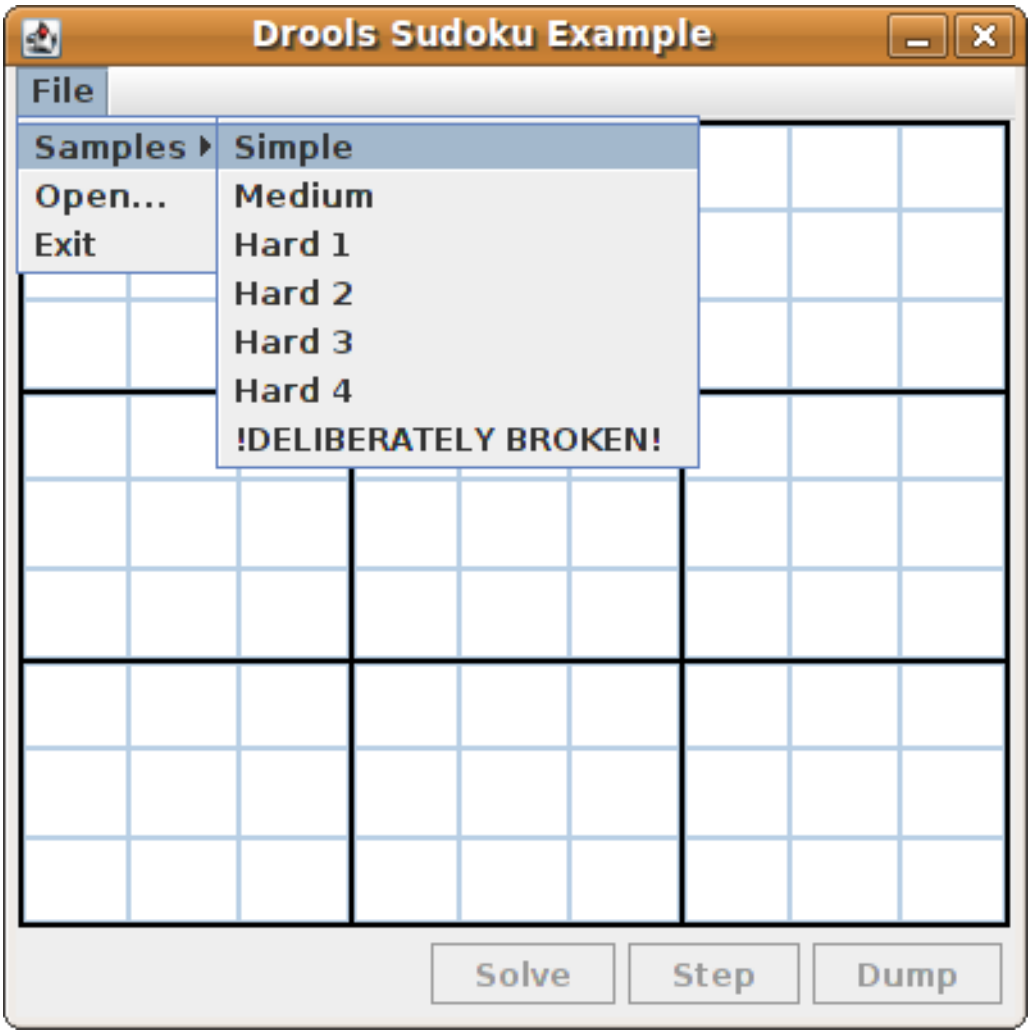
数独例の実行および対話

他の Red Hat Process Automation Manager のデシジョン例と同じように、お使いの IDE で **org.drools.examples.sudoku.SudokuExample** クラスを Java アプリケーションとして実行し、数独の例を実行します。

数独の例を実行すると、**Drools Sudoku Example** GUI ウィンドウが表示されますこのウィンドウには、空白のグリッドが含まれますが、プログラムには、読み込みや解決が可能なグリッドが複数、内部に格納されています。

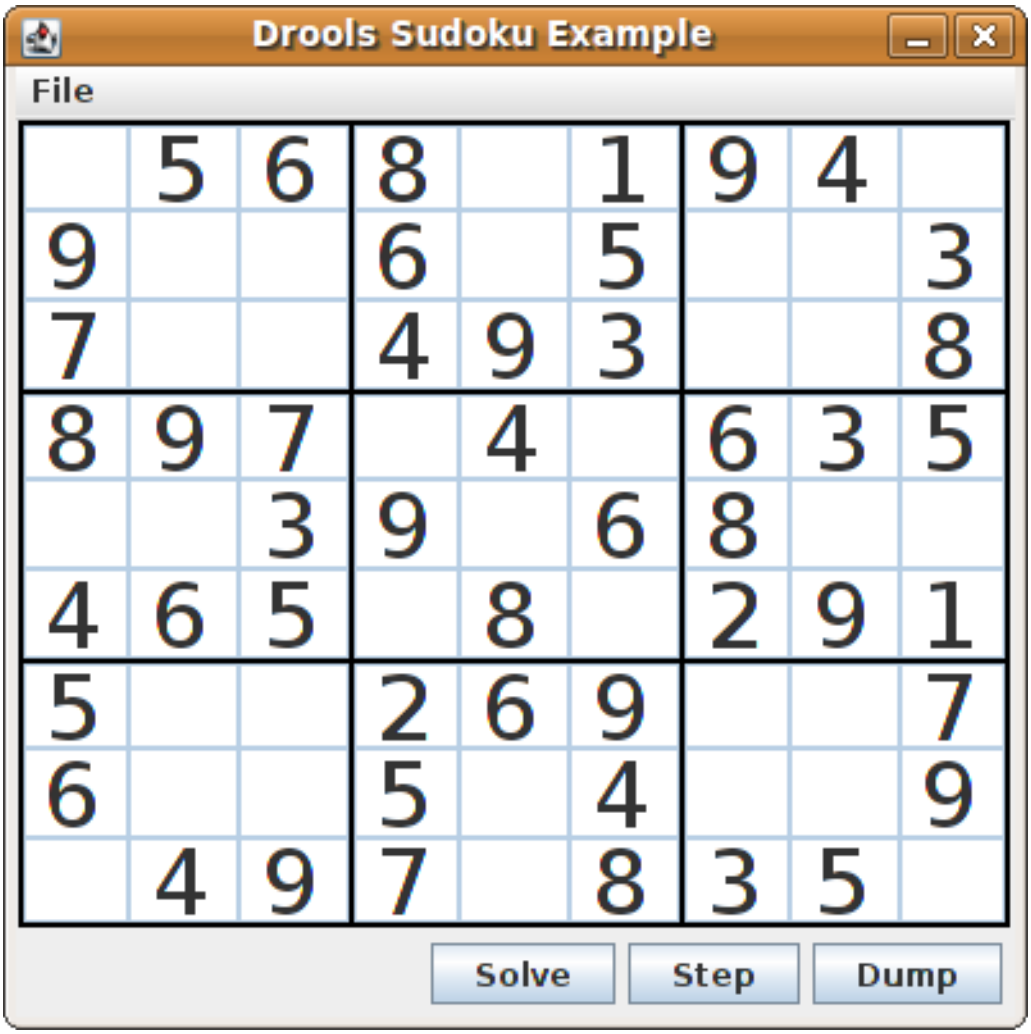
File → **Samples** → **Simple** をクリックして、例の 1 つを読み込みます。グリッドが読み込まれるまで、すべてのボタンが無効になっている点に注目してください。

図7.15 起動後の数独例の GUI



Simple サンプルを読み込むと、パズルの最初の状態に合わせて、グリッドが埋められます。

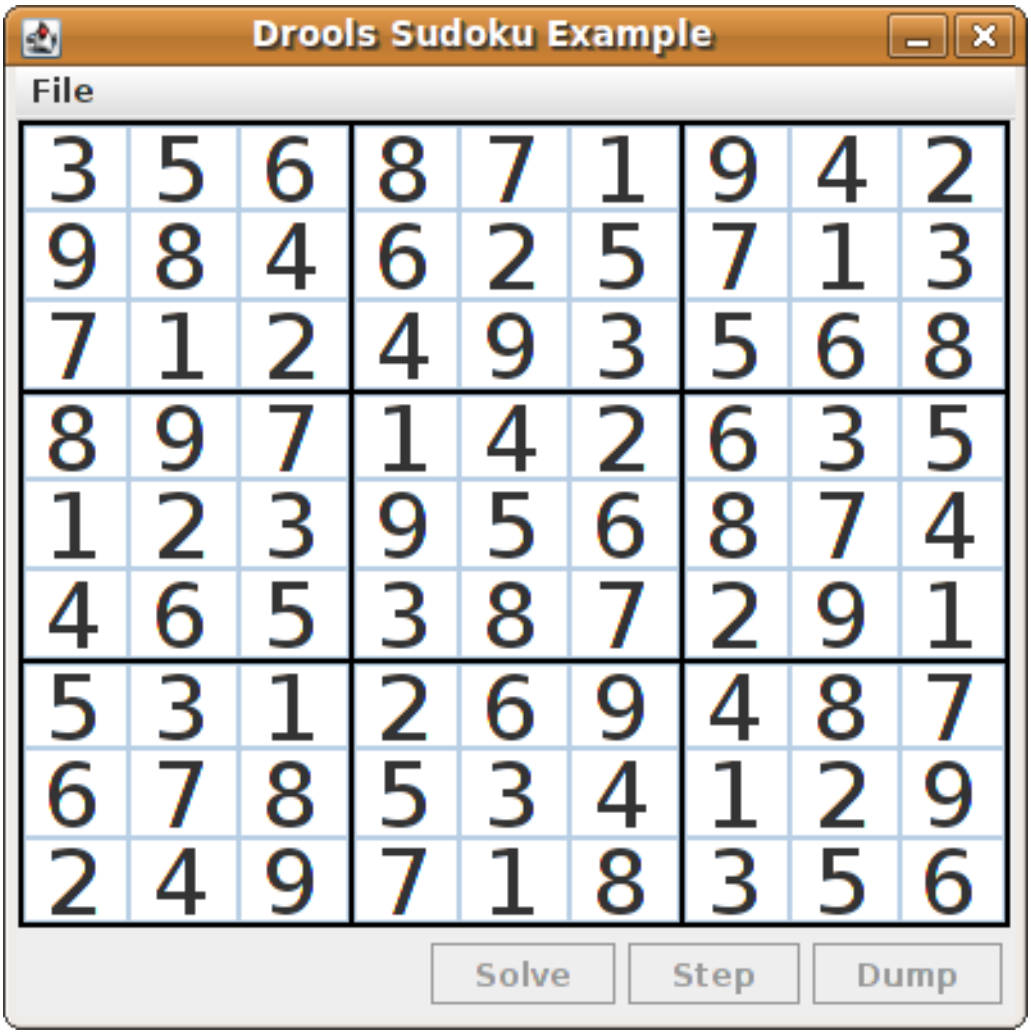
図7.16 Simple サンプルを読み込んだ後の数独例の GUI



以下のオプションから選択します。

- **Solve** をクリックして、数独の例に定義されているルールを実行し、残りの値を埋めていき、このボタンを再度無効にします。

図7.17 Simple サンプルの解決



- **Step** をクリックして、ルールセットに含まれる次の数字を表示します。IDE のコンソールウィンドウでは、解決手順を実行するルールに関する情報が詳細に表示されます。

IDE コンソールでの手順実行の出力

```
single 8 at [0,1]
column elimination due to [1,2]: remove 9 from [4,2]
hidden single 9 at [1,2]
row elimination due to [2,8]: remove 7 from [2,4]
remove 6 from [3,8] due to naked pair at [3,2] and [3,7]
hidden pair in row at [4,6] and [4,4]
```

- **Dump** をクリックしてグリッドの状態を表示します。セルには、解決済みの値か、残りの候補値が表示されます。

IDE コンソールでのダンプ実行の出力

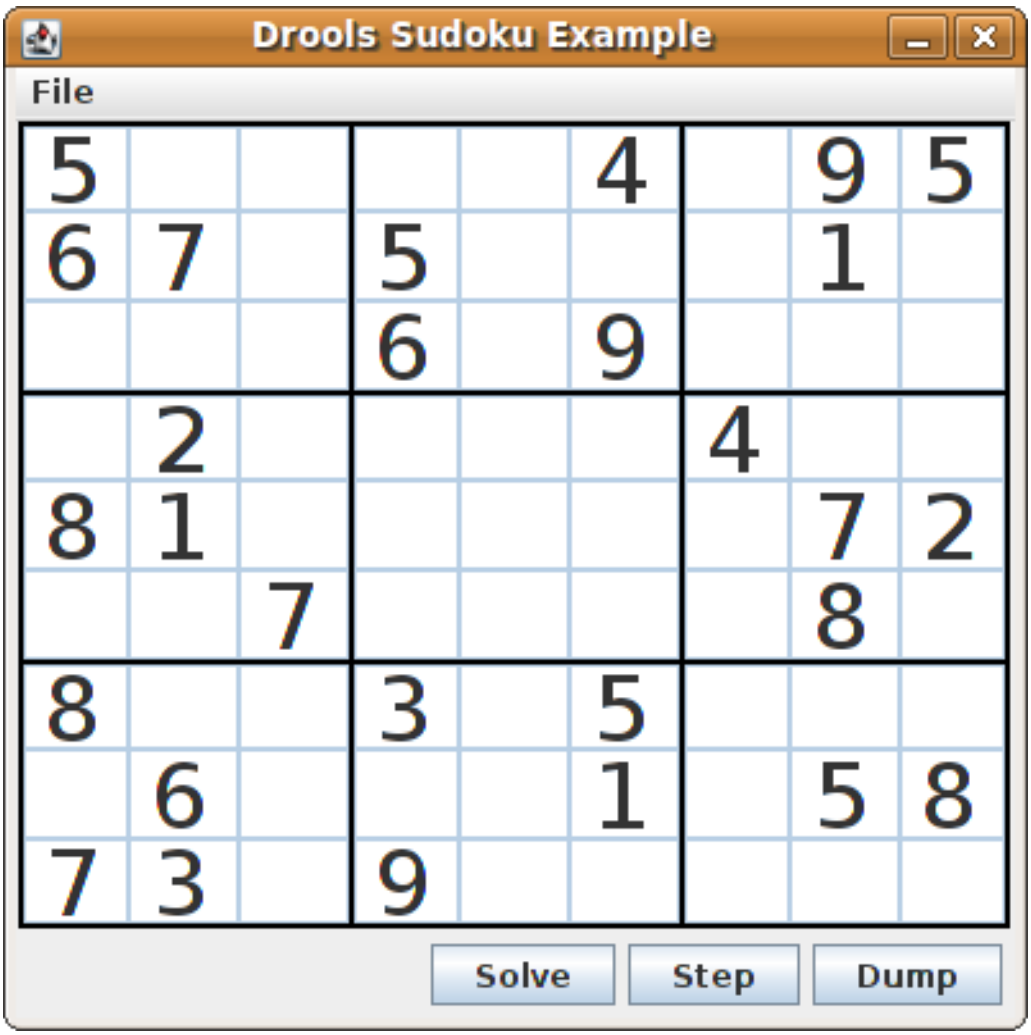
```
        Col: 0     Col: 1     Col: 2     Col: 3     Col: 4     Col:
5     Col: 6     Col: 7     Col: 8
Row 0:  123456789  --- 5 ---  --- 6 ---  --- 8 ---  123456789  ---
1 ---  --- 9 ---  --- 4 ---  123456789
Row 1:  --- 9 ---  123456789  123456789  --- 6 ---  123456789  ---
5 ---  123456789  123456789  --- 3 ---
Row 2:  --- 7 ---  123456789  123456789  --- 4 ---  --- 9 ---  ---
```

```
3 ---  123456789  123456789  --- 8 ---
Row 3:  --- 8 ---  --- 9 ---  --- 7 ---  123456789  --- 4 ---
123456789  --- 6 ---  --- 3 ---  --- 5 ---
Row 4:  123456789  123456789  --- 3 ---  --- 9 ---  123456789  ---
6 ---  --- 8 ---  123456789  123456789
Row 5:  --- 4 ---  --- 6 ---  --- 5 ---  123456789  --- 8 ---
123456789  --- 2 ---  --- 9 ---  --- 1 ---
Row 6:  --- 5 ---  123456789  123456789  --- 2 ---  --- 6 ---  ---
9 ---  123456789  123456789  --- 7 ---
Row 7:  --- 6 ---  123456789  123456789  --- 5 ---  123456789  ---
4 ---  123456789  123456789  --- 9 ---
Row 8:  123456789  --- 4 ---  --- 9 ---  --- 7 ---  123456789  ---
8 ---  --- 3 ---  --- 5 ---  123456789
```

数独の例には、不備のあるサンプルファイルが意図的に含められています。このファイルは、例で定義したルールを使用して解決できます。

File → **Samples** → **!DELIBERATELY BROKEN!** をクリックして、不備のあるサンプルを読み込みます。グリッドは、最初の行に **5** の値を 2 回表示できないにもかかわらず、表示されるなど、問題が含まれた状態で表示されます。

図7.18 不備のある数独例の最初の状態



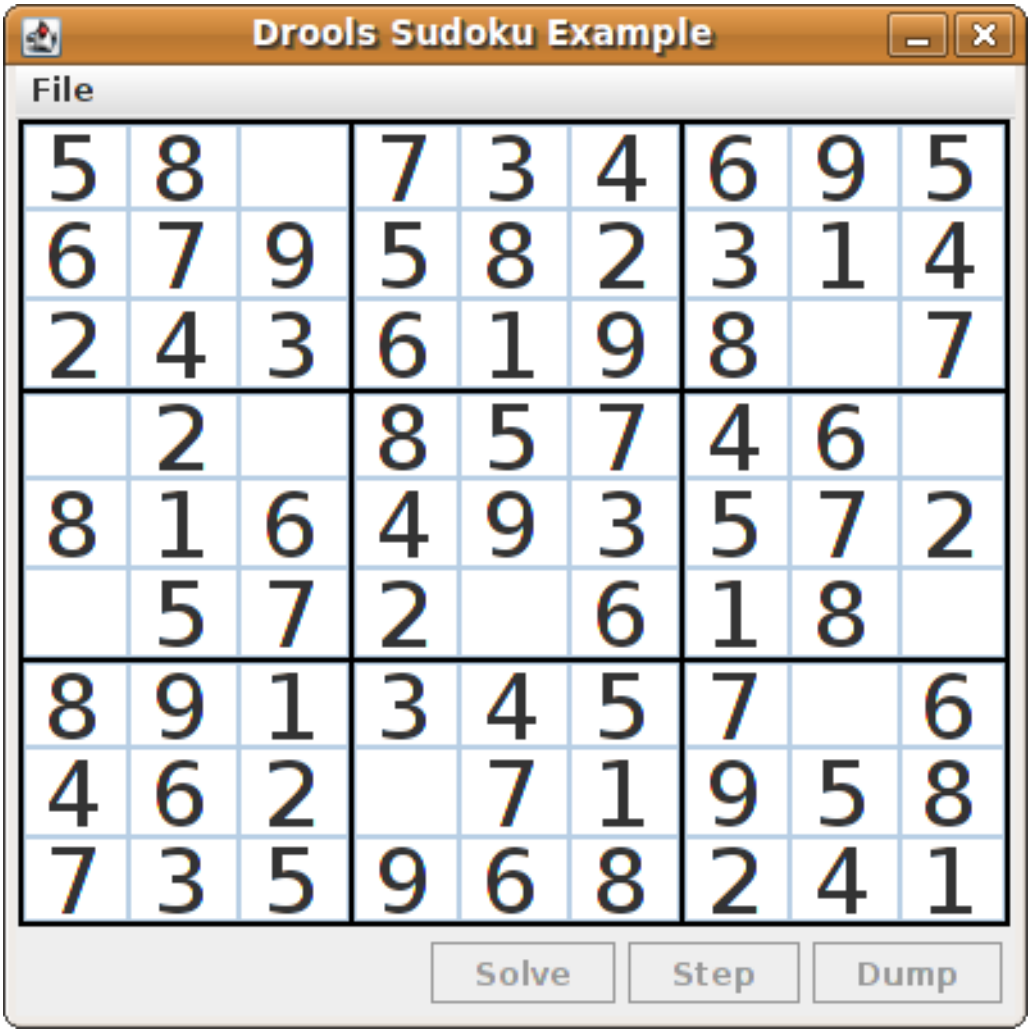
Solve をクリックしてこの無効なグリッドに解決ルールを適用します。数独の例に含まれる関連の解決ルールにより、サンプルの問題が検出され、できる限りパズル解決します。このプロセスでは、すべてを完了させず、空白のセルをいくつか残します。

解決ルールのアクティビティーが IDE コンソールウィンドウに表示されます。

不備のあるサンプルでの問題検出

```
cell [0,8]: 5 has a duplicate in row 0
cell [0,0]: 5 has a duplicate in row 0
cell [6,0]: 8 has a duplicate in col 0
cell [4,0]: 8 has a duplicate in col 0
Validation complete.
```

図7.19 不備のあるサンプルの解決試行



Hard のラベルの付いた数独サンプルファイルはより複雑で、解決ルールを使用しても解決できない可能性があります。解決をしようとして失敗した場合には、IDE コンソールウィンドウに表示されます。

解決不可の Hard サンプル

```
Validation complete.
...
Sorry - can't solve this grid.
```

不備のあるサンプルを解決するためのルールでは、セルの候補となりえる値をもとにした標準の解決手法を実装します。たとえば、セットに値が 1 つ含まれる場合には、これが値になります。セルが 9 個あるグループの 1 つに値が 1 度挿入された場合に、ルールを使用して、特定のセルに対する値を持ち、タイプが **`Setting`** のファクトを挿入します。このファクトは、セルが含まれるグループの他のセルすべてからこの値を取り除き、この値を (選択肢から) 取り消します。

この例の他のルールで、セルに入力可能な値を減らしていきます。**"naked pair"**、**"hidden pair in row"**、**"hidden pair in column"** および **"hidden pair in square"** のルールでは、候補の絞り込みはできますが、回答を得ることはできません。**"X-wings in rows"**、"`X-wings in columns"`、**"intersection removal row"** および **"intersection removal column"** のルールは、より精緻な絞り込みを実行します。

数独例のクラス

org.drools.examples.sudoku.swing パッケージには、以下のように、数独パズルのフレームワークを実装する主なクラスセットが含まれます。

- **SudokuGridModel** は、9x9 グリッドの **Cell** オブジェクトとして数独パズルを格納するために実装可能なインターフェースを定義しています。
- **SudokuGridView** クラスは Swing コンポーネントで **SudokuGridModel** クラス実装の視覚化が可能です。
- **SudokuGridEvent** および **SudokuGridListener** クラスは、モデルとビューの間のステータスの変化をやり取りするために使用します。セルの値が解決または変更されると、イベントが実行されます。
- **SudokuGridSamples** クラスは、デモ目的に一部入力されている数独パズルを複数提供します。

注記

このパッケージには、Red Hat Process Automation Manager ライブラリーの依存関係は含まれません。

org.drools.examples.sudoku パッケージには、以下のように、基本的な **Cell** オブジェクトと各種アグリゲーションを実装する主なクラスセットが含まれます。

- **CellRow**、**CellCol** および **CellSqr**のサブクラスを含む **CellFile** クラス。これらすべては、**CellGroup** クラスのサブタイプになります。
- **Cell** と **CellGroup** は **SetOfNine** のサブクラスで、**Set<Integer>** 型の **free** プロパティーを提供します。**Cell** クラスは、個別の候補セットを表します。**CellGroup** は、セルの全候補セットの統合 (割り当ての必要のある数値セット) です。
  数独の例には、81 個の **Cell** と 27 個の **CellGroup** オブジェクト、 **Cell** プロパティーの **cellRow**、**cellCol** および **cellSqr** が提供するリンク、**CellGroup** プロパティー **cells** (**Cell** オブジェクトリスト) が提供するリストが含まれます。これらのコンポーネントを使用して、セルに値を割り当てたり、候補セットから値を取り除いたりできるように、特定の状態を検出するルールを記述できます。
- **Setting** クラスを使用して、値の割り当てに伴うオペレーションをトリガーします。**Setting** ファクトは、整合性の取れない中間の状態に対して反応しないように、新しい状況を検出する全ルールに配置して使用します。
- **Stepping** クラスは、優先順位が低いルールに使用して、**"Step"** が予期なく中断された場合に緊急停止を行います。この動作は、プログラムでパズルを解決できないということです。
- Main クラス **org.drools.examples.sudoku.SudokuExample** は、全コンポーネントを統合する Java アプリケーションを実装します。

数独の検証ルール (validate.drl)

数独の例の **validate.drl** ファイルには、セルグループで数が重複している状況を検出する検証ルー

ルが含まれます。このグループは、**"validate"** アジェンダグループに統合され、ユーザーがパズルを読み込むと、明示的にルールをアクティベートできます。

"duplicate in cell …" の 3 つのルールの **when** 条件はすべて以下の方法で機能します。

- このルールの最初の条件で、割り当てられた値でセルを特定します。
- このルールの 2 番目の条件では、3 つのセルグループのどれかを所属先にプルします。
- 最終条件は、ルールに従い、最初のセル、同じ行、列、または四角に入る値と同じセル (上記のセル以外) を検索します。

ルール "duplicate in cell …"

```
rule "duplicate in cell row"
  when
    $c: Cell( $v: value != null )
    $cr: CellRow( cells contains $c )
    exists Cell( this != $c, value == $v, cellRow == $cr )
  then
    System.out.println( "cell " + $c.toString() + " has a duplicate in row
" + $cr.getNumber() );
end

rule "duplicate in cell col"
  when
    $c: Cell( $v: value != null )
    $cc: CellCol( cells contains $c )
    exists Cell( this != $c, value == $v, cellCol == $cc )
  then
    System.out.println( "cell " + $c.toString() + " has a duplicate in col
" + $cc.getNumber() );
end

rule "duplicate in cell sqr"
  when
    $c: Cell( $v: value != null )
    $cs: CellSqr( cells contains $c )
    exists Cell( this != $c, value == $v, cellSqr == $cs )
  then
    System.out.println( "cell " + $c.toString() + " has duplicate in its
square of nine." );
end
```

ルール **"terminate group"** は最後に実行されます。このルールは、メッセージを出力して、シーケンスを停止します。

ルール "terminate group"

```
rule "terminate group"
    salience -100
  when
  then
    System.out.println( "Validation complete." );
    drools.halt();
end
```

■

数独の解決ルール (sudoku.drl)

数独の例の **sudoku.drl** ファイルには、3 種類の ルールタイプが含まれます。1 つ目のグループは、セルへの数値の割り当てを処理して、もう 1 つは実行可能な割り当てを検出して、3 つ目は候補セットからの値を削除します。

"set a value"、**"eliminate a value from Cell"** および **"retract setting"** のルールは、**Setting** オブジェクトの有無により左右されます。最初のルールは、セルへの割り当てと、3 つのセルグループの **free** セットから値を削除する操作を処理します。また、ゼロの場合には、このグループはカウンターを 1 つ減らし、**fireUntilHalt()** を呼び出した Java アプリケーションに制御を戻します。

"eliminate a value from Cell" ルールの目的は、新たに割り当てられたセルに関連する全セルの候補リストを絞り込むことです。最後に、すべての除外が完了したら、**"retract setting"** ルールにより、トリガーされている **Setting** ファクトを取り消します。

ルール "set a value"、"eliminate a value from a Cell" および "retract setting"

```
// A Setting object is inserted to define the value of a Cell.
// Rule for updating the cell and all cell groups that contain it
rule "set a value"
  when
    // A Setting with row and column number, and a value
    $s: Setting( $rn: rowNo, $cn: colNo, $v: value )

    // A matching Cell, with no value set
    $c: Cell( rowNo == $rn, colNo == $cn, value == null,
              $cr: cellRow, $cc: cellCol, $cs: cellSqr )

    // Count down
    $ctr: Counter( $count: count )
  then
    // Modify the Cell by setting its value.
    modify( $c ){ setValue( $v ) }
    // System.out.println( "set cell " + $c.toString() );
    modify( $cr ){ blockValue( $v ) }
    modify( $cc ){ blockValue( $v ) }
    modify( $cs ){ blockValue( $v ) }
    modify( $ctr ){ setCount( $count - 1 ) }
end

// Rule for removing a value from all cells that are siblings
// in one of the three cell groups
rule "eliminate a value from Cell"
  when
    // A Setting with row and column number, and a value
    $s: Setting( $rn: rowNo, $cn: colNo, $v: value )

    // The matching Cell, with the value already set
    Cell( rowNo == $rn, colNo == $cn, value == $v, $exCells: exCells )

    // For all Cells that are associated with the updated cell
    $c: Cell( free contains $v ) from $exCells
  then
    // System.out.println( "clear " + $v + " from cell " +
```

```
$c.posAsString()  );
    // Modify a related Cell by blocking the assigned value.
    modify( $c ){ blockValue( $v ) }
end

// Rule for eliminating the Setting fact
rule "retract setting"
  when
    // A Setting with row and column number, and a value
    $s: Setting( $rn: rowNo, $cn: colNo, $v: value )

    // The matching Cell, with the value already set
    $c: Cell( rowNo == $rn, colNo == $cn, value == $v )

    // This is the negation of the last pattern in the previous rule.
    // Now the Setting fact can be safely retracted.
    not( $x: Cell( free contains $v )
         and
         Cell( this == $c, exCells contains $x ) )
  then
    // System.out.println( "done setting cell " + $c.toString() );
    // Discard the Setter fact.
    delete( $s );
    // Sudoku.sudoku.consistencyCheck();
end
```

解決ルール 2 つを使用して、セルに数字を割り当てることができる状況を検出します。**"single"** のルールは、**Cell** に、数字が 1 つだけの候補セットが含まれる場合に実行されます。**"hidden single"** ルールは、候補が 1 つだけのセルが存在しない場合に実行されますが、セルに候補が含まれる場合には、セルが所属する 3 つのグループの 1 つに含まれるその他すべてのセルに、この候補が存在しないということです。いずれのルールも **Setting** ファクトを作成して、挿入します。

ルール "single" および "hidden single"

```
// Detect a set of candidate values with cardinality 1 for some Cell.
// This is the value to be set.
rule "single"
  when
    // Currently no setting underway
    not Setting()

    // One element in the "free" set
    $c: Cell( $rn: rowNo, $cn: colNo, freeCount == 1 )
  then
    Integer i = $c.getFreeValue();
    if (explain) System.out.println( "single " + i + " at " +
$c.posAsString() );
    // Insert another Setter fact.
    insert( new Setting( $rn, $cn, i ) );
end

// Detect a set of candidate values with a value that is the only one
// in one of its groups. This is the value to be set.
rule "hidden single"
  when
    // Currently no setting underway
```

```
    not Setting()
    not Cell( freeCount == 1 )

    // Some integer
    $i: Integer()

    // The "free" set contains this number
    $c: Cell( $rn: rowNo, $cn: colNo, freeCount > 1, free contains $i )

    // A cell group contains this cell $c.
    $cg: CellGroup( cells contains $c )
    // No other cell from that group contains $i.
    not ( Cell( this != $c, free contains $i ) from $cg.getCells() )
  then
    if (explain) System.out.println( "hidden single " + $i + " at " +
$c.posAsString() );
    // Insert another Setter fact.
    insert( new Setting( $rn, $cn, $i ) );
end
```

最大グループからのルール (個別または 2 - 3 のグループ単位) は、数独パズルを手作業で解決するのに使用する、さまざまな解決手法を実装します。

"naked pair" ルールは、グループの 2 つのセルで、全く同じ候補セットでサイズ **2** のものを検出します。これらの 2 つの値は、対象グループの他の候補セットすべてから削除することができます。

ルール "naked pair"

```
// A "naked pair" is two cells in some cell group with their sets of
// permissible values being equal with cardinality 2. These two values
// can be removed from all other candidate lists in the group.
rule "naked pair"
  when
    // Currently no setting underway
    not Setting()
    not Cell( freeCount == 1 )

    // One cell with two candidates
    $c1: Cell( freeCount == 2, $f1: free, $r1: cellRow, $rn1: rowNo, $cn1:
colNo, $b1: cellSqr )

    // The containing cell group
    $cg: CellGroup( freeCount > 2, cells contains $c1 )

    // Another cell with two candidates, not the one we already have
    $c2: Cell( this != $c1, free == $f1 /*** , rowNo >= $rn1, colNo >=
$cn1 ***/ ) from $cg.cells

    // Get one of the "naked pair".
    Integer( $v: intValue ) from $c1.getFree()

    // Get some other cell with a candidate equal to one from the pair.
    $c3: Cell( this != $c1 && != $c2, freeCount > 1, free contains $v )
from $cg.cells
  then
    if (explain) System.out.println( "remove " + $v + " from " +
```

```
$c3.posAsString() + " due to naked pair at " + $c1.posAsString() + " and "
+ $c2.posAsString() );
    // Remove the value.
    modify( $c3 ){ blockValue( $v ) }
end
```

3 番目のルール **"hidden pair in …"** は、ルール **"naked pair"** と同様に機能します。ルールはグループの 2 つのセルで 2 つの数字を検出します。どの値もこのグループの他のセルには入りません。つまり、他の候補はすべて、隠れたペアを持つ 2 つのセルから削除します。

ルール "hidden pair in …"

```
// If two cells within the same cell group contain candidate sets with
more than
// two values, with two values being in both of them but in none of the
other
// cells, then we have a "hidden pair". We can remove all other candidates
from
// these two cells.
rule "hidden pair in row"
  when
    // Currently no setting underway
    not Setting()
    not Cell( freeCount == 1 )

    // Establish a pair of Integer facts.
    $i1: Integer()
    $i2: Integer( this > $i1 )

    // Look for a Cell with these two among its candidates. (The upper
bound on
    // the number of candidates avoids a lot of useless work during
startup.)
    $c1: Cell( $rn1: rowNo, $cn1: colNo, freeCount > 2 && < 9, free
contains $i1 && contains $i2, $cellRow: cellRow )

    // Get another one from the same row, with the same pair among its
candidates.
    $c2: Cell( this != $c1, cellRow == $cellRow, freeCount > 2, free
contains $i1 && contains $i2 )

    // Ascertain that no other cell in the group has one of these two
values.
    not( Cell( this != $c1 && != $c2, free contains $i1 || contains $i2 )
from $cellRow.getCells() )
  then
    if( explain) System.out.println( "hidden pair in row at " +
$c1.posAsString() + " and " + $c2.posAsString() );
    // Set the candidate lists of these two Cells to the "hidden pair".
    modify( $c1 ){ blockExcept( $i1, $i2 ) }
    modify( $c2 ){ blockExcept( $i1, $i2 ) }
end

rule "hidden pair in column"
  when
    not Setting()
```

```
    not Cell( freeCount == 1 )

    $i1: Integer()
    $i2: Integer( this > $i1 )
    $c1: Cell( $rn1: rowNo, $cn1: colNo, freeCount > 2 && < 9, free
contains $i1 && contains $i2, $cellCol: cellCol )
    $c2: Cell( this != $c1, cellCol == $cellCol, freeCount > 2, free
contains $i1 && contains $i2 )
    not( Cell( this != $c1 && != $c2, free contains $i1 || contains $i2 )
from $cellCol.getCells() )
  then
    if (explain) System.out.println( "hidden pair in column at " +
$c1.posAsString() + " and " + $c2.posAsString() );
    modify( $c1 ){ blockExcept( $i1, $i2 ) }
    modify( $c2 ){ blockExcept( $i1, $i2 ) }
end

rule "hidden pair in square"
  when
    not Setting()
    not Cell( freeCount == 1 )

    $i1: Integer()
    $i2: Integer( this > $i1 )
    $c1: Cell( $rn1: rowNo, $cn1: colNo, freeCount > 2 && < 9, free
contains $i1 && contains $i2,
               $cellSqr: cellSqr )
    $c2: Cell( this != $c1, cellSqr == $cellSqr, freeCount > 2, free
contains $i1 && contains $i2 )
    not( Cell( this != $c1 && != $c2, free contains $i1 || contains $i2 )
from $cellSqr.getCells() )
  then
    if (explain) System.out.println( "hidden pair in square " +
$c1.posAsString() + " and " + $c2.posAsString() );
    modify( $c1 ){ blockExcept( $i1, $i2 ) }
    modify( $c2 ){ blockExcept( $i1, $i2 ) }
end
```

2 つのルールは行と列で **"X-wings"** を処理します。2 つの異なる行 (または列) で、ある値を入力できるセルが 2 つしかなく、これらの候補が同じ列 (または行) に入る場合に、この列 (または行) のこの値に対する他の候補は除外できます。これらのルールの 1 つに含まれるパターンシーケンスに従うと、**same** または **only** などの用語で都合よく表現されている条件は、適切な制約が付けられたパターンになるか、**not** の接頭辞が付きます。

ルール "X-wings in ..."

```
rule "X-wings in rows"
  when
    not Setting()
    not Cell( freeCount == 1 )

    $i: Integer()
    $ca1: Cell( freeCount > 1, free contains $i,
                $ra: cellRow, $rano: rowNo,         $c1: cellCol,
$c1no: colNo )
    $cb1: Cell( freeCount > 1, free contains $i,
```

```
                $rb: cellRow, $rbno: rowNo > $rano,      cellCol == $c1 )
    not( Cell( this != $ca1 && != $cb1, free contains $i ) from
$c1.getCells() )

    $ca2: Cell( freeCount > 1, free contains $i,
                cellRow == $ra, $c2: cellCol,       $c2no: colNo > $c1no
)
    $cb2: Cell( freeCount > 1, free contains $i,
                cellRow == $rb,      cellCol == $c2 )
    not( Cell( this != $ca2 && != $cb2, free contains $i ) from
$c2.getCells() )

    $cx: Cell( rowNo == $rano || == $rbno, colNo != $c1no && != $c2no,
               freeCount > 1, free contains $i )
  then
    if (explain) {
        System.out.println( "X-wing with " + $i + " in rows " +
            $ca1.posAsString() + " - " + $cb1.posAsString() +
            $ca2.posAsString() + " - " + $cb2.posAsString() + ", remove
from " + $cx.posAsString() );
    }
    modify( $cx ){ blockValue( $i ) }
end

rule "X-wings in columns"
  when
    not Setting()
    not Cell( freeCount == 1 )

    $i: Integer()
    $ca1: Cell( freeCount > 1, free contains $i,
                $c1: cellCol, $c1no: colNo,         $ra: cellRow,
$rano: rowNo )
    $ca2: Cell( freeCount > 1, free contains $i,
                $c2: cellCol, $c2no: colNo > $c1no,      cellRow == $ra )
    not( Cell( this != $ca1 && != $ca2, free contains $i ) from
$ra.getCells() )

    $cb1: Cell( freeCount > 1, free contains $i,
                cellCol == $c1, $rb: cellRow,  $rbno: rowNo > $rano )
    $cb2: Cell( freeCount > 1, free contains $i,
                cellCol == $c2,      cellRow == $rb )
    not( Cell( this != $cb1 && != $cb2, free contains $i ) from
$rb.getCells() )

    $cx: Cell( colNo == $c1no || == $c2no, rowNo != $rano && != $rbno,
               freeCount > 1, free contains $i )
  then
    if (explain) {
        System.out.println( "X-wing with " + $i + " in columns " +
            $ca1.posAsString() + " - " + $ca2.posAsString() +
            $cb1.posAsString() + " - " + $cb2.posAsString() + ", remove
from " + $cx.posAsString()  );
    }
    modify( $cx ){ blockValue( $i ) }
end
```

"intersection removal …" の 2 つのルールは、1 つの四角の中に (1 つの行または列) 使用できる数字を制限するというルールに基づいています。つまり、この番号は行または列の中の 2-3 セルの 1 つに入っていないといけないのです。グループの別のセルすべての中にある候補セットから削除できます。このパターンは、発生制限を確立して、同じセルファイルの中かつ、四角の外のセルそれぞれに対して実行されます。

ルール "intersection removal …"

```
rule "intersection removal column"
  when
    not Setting()
    not Cell( freeCount == 1 )

    $i: Integer()
    // Occurs in a Cell
    $c: Cell( free contains $i, $cs: cellSqr, $cc: cellCol )
    // Does not occur in another cell of the same square and a different
column
    not Cell( this != $c, free contains $i, cellSqr == $cs, cellCol != $cc
)

    // A cell exists in the same column and another square containing this
value.
    $cx: Cell( freeCount > 1, free contains $i, cellCol == $cc, cellSqr !=
$cs )
  then
    // Remove the value from that other cell.
    if (explain) {
        System.out.println( "column elimination due to " +
$c.posAsString() +
                            ": remove " + $i + " from " +
$cx.posAsString() );
    }
    modify( $cx ){ blockValue( $i ) }
end

rule "intersection removal row"
  when
    not Setting()
    not Cell( freeCount == 1 )

    $i: Integer()
    // Occurs in a Cell
    $c: Cell( free contains $i, $cs: cellSqr, $cr: cellRow )
    // Does not occur in another cell of the same square and a different
row.
    not Cell( this != $c, free contains $i, cellSqr == $cs, cellRow != $cr
)

    // A cell exists in the same row and another square containing this
value.
    $cx: Cell( freeCount > 1, free contains $i, cellRow == $cr, cellSqr !=
$cs )
  then
    // Remove the value from that other cell.
    if (explain) {
```

```
        System.out.println( "row elimination due to " + $c.posAsString() +
                            ": remove " + $i + " from " +
$cx.posAsString() );
    }
    modify( $cx ){ blockValue( $i ) }
end
```

これらのルールは、すべてではありませんが、多くの数独パズルでは十分です。非常に難度の高いグリッドを解決するには、ルールセットにはさらに複雑なルールが必要です (最終的には、パズルは試行錯誤でしか解決できません)。

7.8. CONWAY の GAME OF LIFE 例のデシジョン (ルールフローグループおよび GUI 統合)

John Conway による有名なセルオートマトン (CA: Cellular automation) をベースにした Conway の Game of Life 例のデシジョンセットは、ルールでルールフローグループを使用してルール実行を制御する方法を例示します。またこの例は、Red Hat Process Automation Manager ルールをグラフィカルユーザーインターフェース (GUI) と統合する方法も例示しています。今回は、Conway の Game of Life を Swing ベースで実装しています。

以下は、Conway の Game of Life の例の概要です。

- **名前**: **conway**
- **Main クラス**: (**src/main/java** 内の) **org.drools.examples.conway.ConwayRuleFlowGroupRun**、**org.drools.examples.conway.ConwayAgendaGroupRun**
- **モジュール**: **droolsjbpm-integration-examples**
- **タイプ**: Java アプリケーション
- **ルールファイル**: (**src/main/resources** 内の) **org.drools.examples.conway.*.drl**
- **目的**: ルールフローグループと GUI 統合を例示します。

注記

Conway の Game of Life の例は、Red Hat Process Automation Manager に含まれる、他の例のデシジョンセットの多くとは異なり、Red Hat カスタマーポータル から取得する **Red Hat Process Automation Manager 7.2.0 Source Distribution** の **~/rhpam-7.2.0-sources/src/droolsjbpm-integration-$VERSION/droolsjbpm-integration-examples** に配置されています。

Conway の Game of Life では、初期設定または定義済みのプロパティーで高度なパターンを作成して、初期状態からどのように進化していくかを観察することで、ユーザーはゲームと対話します。ゲームの目的は、世代ごとに人口の成長を表示します。各世代は、すべてのセル (細胞) が同時に進化していき、前の世代をもとにして生み出されます。

以下の基本的なルールで、次の世代がどのようになるかを制御していきます。

- 生きているセルの近傍に、生きているセルが 2 個未満の場合は、孤独で死んでしまう。
- 生きているセルの近傍に、生きているセルが 4 個以上ある場合は、過密で死んでしまう。

- 死亡したセルの近傍に、生きているセルがちょうど 3 つある場合には、このセルは生き返る。

この基準のいずれも満たさないセルは、そのまま次の世代に残ります。

Conway の Game of Life の例は、**ruleflow-group** 属性が含まれる Red Hat Process Automation Manager ルールで、ゲームに実装されているパターンを定義します。この例には、アジェンダグループを使用して同じ動作を行うデシジョンセットのバージョンも含まれています。アジェンダグループは、エンジンアジェンダをパーティションして、ルールのグループを実行制御できるようにします。デフォルトでは、すべてのルールがアジェンダグループ **MAIN** に含まれています。ルールに異なるアジェンダグループを指定するには、**agenda-group** 属性を使用できます。

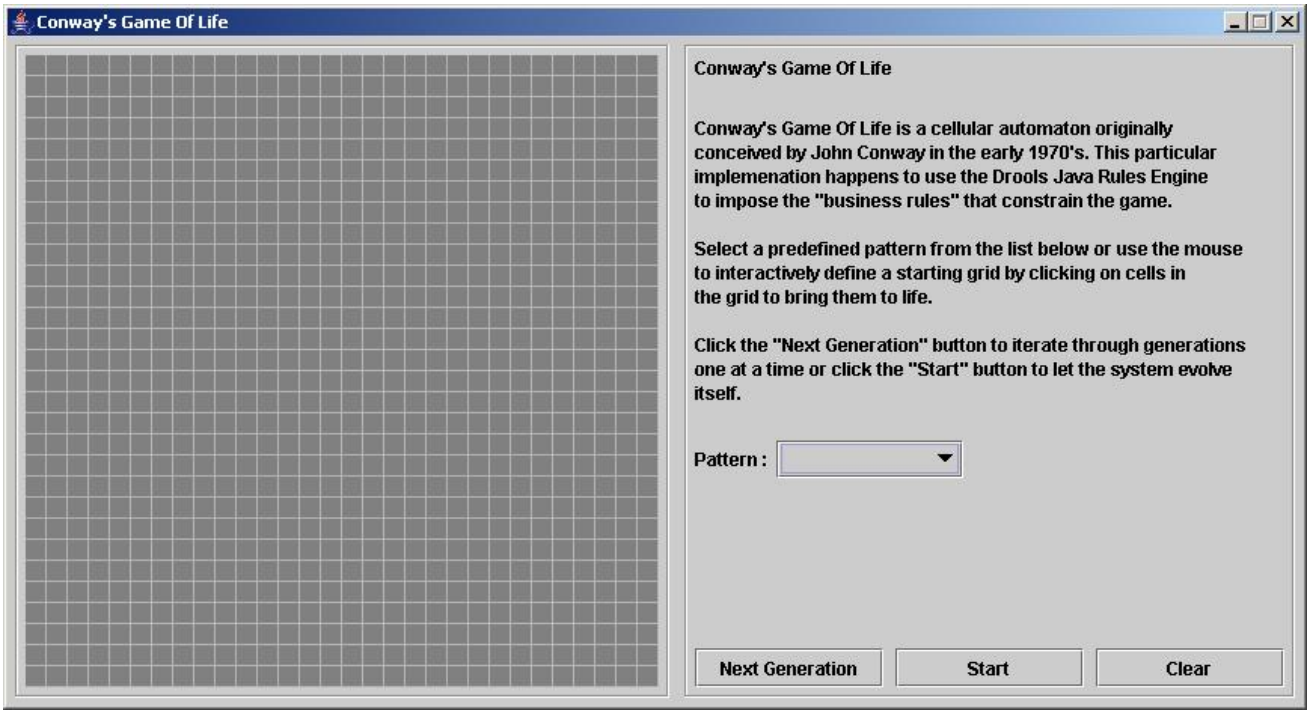
この概要では、Conway の例でアジェンダグループを使用したバージョンは触れません。アジェンダグループの詳細情報は、特にアジェンダグループについて対応している Red Hat Process Automation Manager 例のデシジョンセットを参照してください。

Conway 例の実行および対話

他の Red Hat Process Automation Manager のデシジョン例と同じように、お使いの IDE で **org.drools.examples.conway.ConwayRuleFlowGroupRun** クラスを Java アプリケーションとして実行し、Conway の例を実行します。

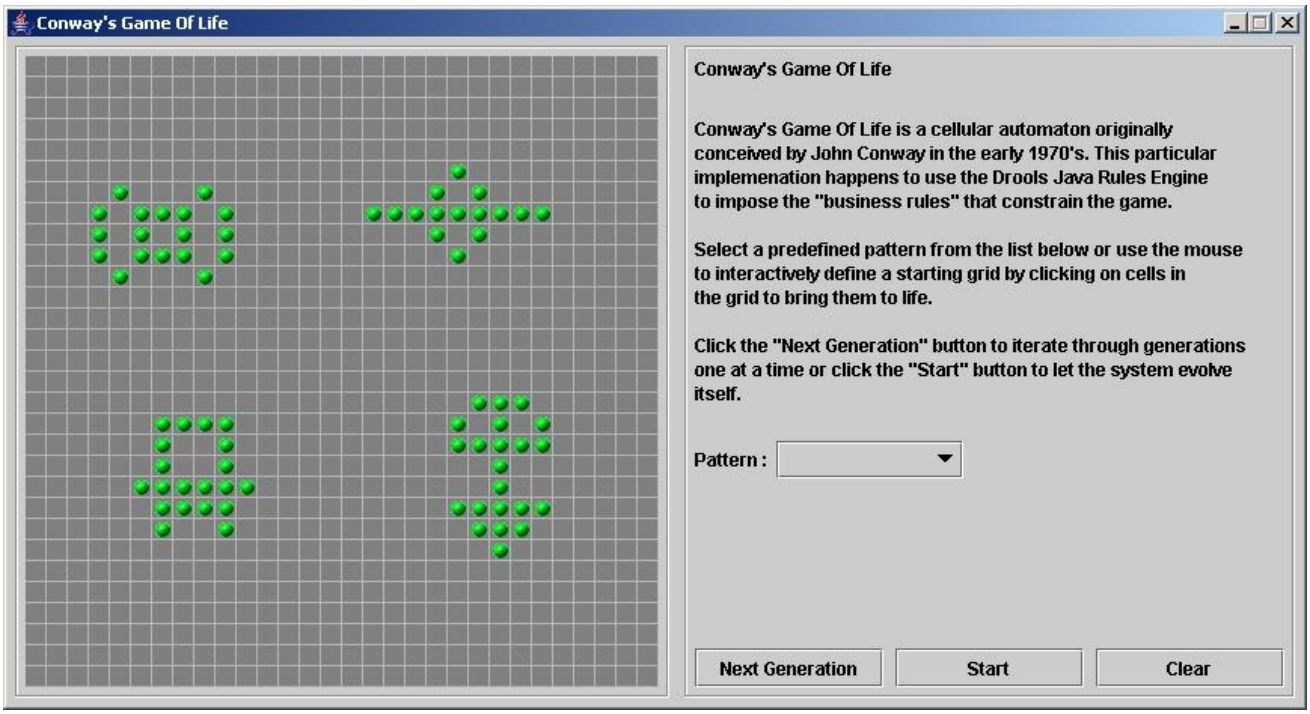
Conway の例を実行すると、**Conway's Game of Life** GUI ウィンドウが表示されます。このウィンドウには、空のグリッドまたは "アリーナ" が含まれており、ここで生命のシミュレーションが行われます。システムにまだ生きているセルが含まれていないので、グリッドは最初は空白です。

図7.20 起動後の Conway 例の GUI



パターン のドロップダウンメニューから事前定義済みのパターンを選択して、**次の世代** をクリックし、各人口の世代をクリックしていきます。セルは生きているか、死んでいるかのどちらかで、生きているセルには緑のボールが含まれます。最初のパターンから人口が進化するにつれ、ゲームのルールをもとに、セルが近傍のセルに合わせて、生存するか、死亡していきます。

図7.21 Conway の例の世代進化



近傍には、上下左右のセルだけでなく対角線上につながっているセルも含まれるので、各セルには合計 8 つの近傍があります。例外は、角のセルと 4 辺上にあるセルで、それぞれ順に近傍が 3 つだけと、5 つだけになります。

セルをクリックすることで手動で介入して、セルを作成することも、死亡させることもできます。

最初のパターンから自動的に進化を実行するには、**スタート** をクリックします。

ルールグループを使用する Conway 例のルール

ConwayRuleFlowGroupRun の例のルールは、ルールフローグループを使用して、ルール実行を制御します。ルールフローグループは、**ruleflow-group** ルール属性に関連付けられたルールのグループです。これらのルールは、このグループがアクティベートされたときにしか実行されません。グループ自体は、ルールフローの図の詳細がグループを表すノードに到達してからでないと、アクティブになりません。

Conway の例では、ルールに以下のルールフローグループを使用します。

- **"register neighbor"**
- **"evaluate"**
- **"calculate"**
- **"reset calculate"**
- **"birth"**
- **"kill"**
- **"kill all"**

Cell オブジェクトはすべて、KIE セッションに挿入され、**"register neighbor"** ルールフローグループの **"register …"** ルールがルールフロープロセスにより実行できるようになります。4 つのルールが含まれるこのグループは、セル同士の **Neighbor** の関係と、北東、北、北西、西の近傍との

Neighbour の関係を作り出します。

この関係は双方向で、他の 4 方向を処理します。各辺上のセルは、特別な対応は必要ありません。これらのセルは、近傍のセルがなければペアは作成されません。

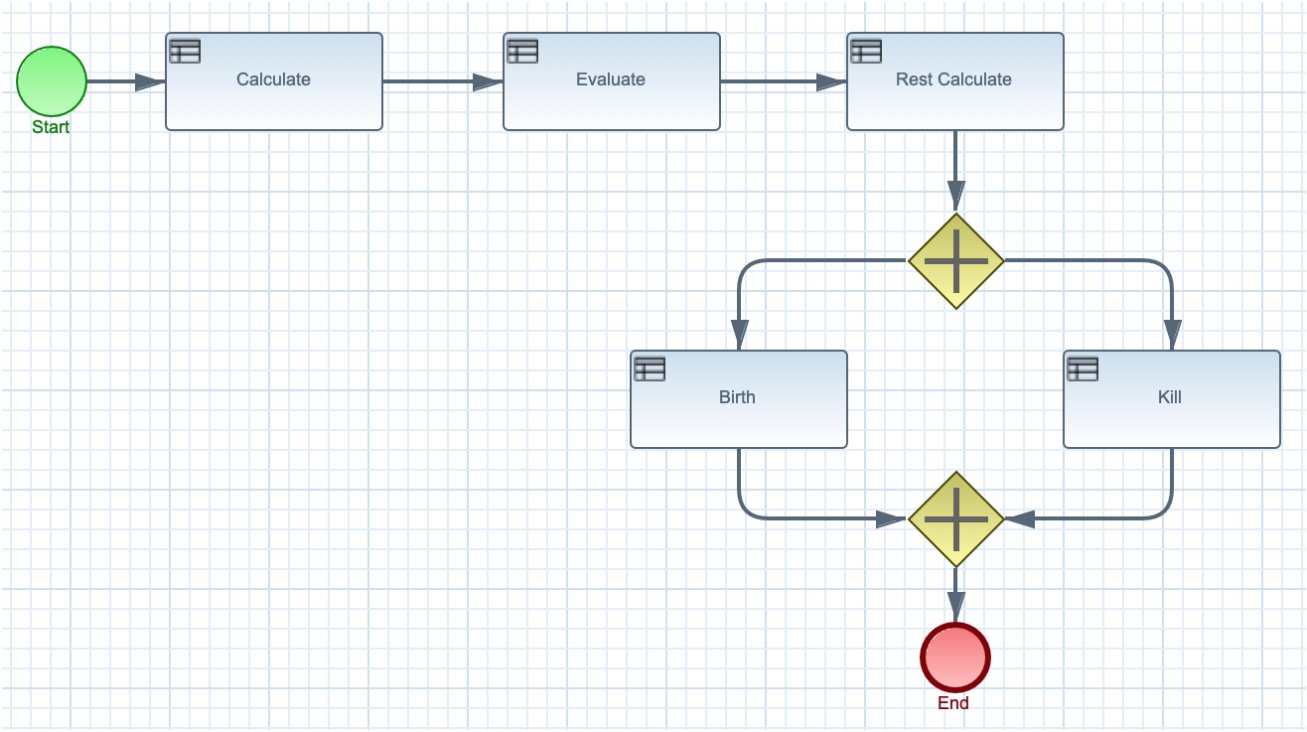
これらのルールに対して、すべてのアクティベーションが実行されるまで、全セルは、近傍の全セルと関係があります。

ルール "register …"

```
rule "register north east"
    ruleflow-group "register neighbor"
  when
    $cell: Cell( $row : row, $col : col )
    $northEast : Cell( row  == ($row - 1), col == ( $col + 1 ) )
  then
    insert( new Neighbor( $cell, $northEast ) );
    insert( new Neighbor( $northEast, $cell ) );
end

rule "register north"
    ruleflow-group "register neighbor"
  when
    $cell: Cell( $row : row, $col : col )
    $north : Cell( row  == ($row - 1), col == $col )
  then
    insert( new Neighbor( $cell, $north ) );
    insert( new Neighbor( $north, $cell ) );
end

rule "register north west"
    ruleflow-group "register neighbor"
  when
    $cell: Cell( $row : row, $col : col )
    $northWest : Cell( row  == ($row - 1), col == ( $col - 1 ) )
  then
    insert( new Neighbor( $cell, $northWest ) );
    insert( new Neighbor( $northWest, $cell ) );
end

rule "register west"
    ruleflow-group "register neighbor"
  when
    $cell: Cell( $row : row, $col : col )
    $west : Cell( row  == $row, col == ( $col - 1 ) )
  then
    insert( new Neighbor( $cell, $west ) );
    insert( new Neighbor( $west, $cell ) );
end
```

全セルが挿入されたら、Java コードはグリッドにパターンを適用し、特定のセルを **Live** に設定します。次に、ユーザーが **スタート** または **次の世代** をクリックすると、**Generation** のルールフローが実行されます。このルールフローは、世代のサイクルごとにセルの変更をすべて管理します。

図7.22 世代のルールフロー



ルールフロープロセスは、実行可能なグループに **"evaluate"** ルールフローグループおよびアクティブなルールを追加します。このグループの **"Kill the …"** と **"Give Birth"** ルールを使用して、細胞の誕生または死亡セルにゲームのルールを適用します。この例では、**phase** 属性を使用して、特定のルールグループで **Cell** オブジェクトの理由付けをトリガーします。通常は、phase はルールフロープロセスの定義に含まれるルールフローグループに紐づけされています。

この例では、変更の適用前に評価を完全に完了しておく必要があるので、この時点では **Cell** オブジェクトの状態は変更されません。細胞の **phase** を **Phase.KILL** または **Phase.BIRTH** に適用し、後ほど **Cell** オブジェクトに適用されたアクションを制御するのに使用します。

ルール "Kill the …" および "Give Birth"

```
rule "Kill The Lonely"
    ruleflow-group "evaluate"
    no-loop
  when
    // A live cell has fewer than 2 live neighbors.
    theCell: Cell( liveNeighbors < 2, cellState == CellState.LIVE,
                   phase == Phase.EVALUATE )
  then
    modify( theCell ){
        setPhase( Phase.KILL );
    }
end

rule "Kill The Overcrowded"
    ruleflow-group "evaluate"
    no-loop
  when
    // A live cell has more than 3 live neighbors.
    theCell: Cell( liveNeighbors > 3, cellState == CellState.LIVE,
                   phase == Phase.EVALUATE )
  then
```

```
    modify( theCell ){
        setPhase( Phase.KILL );
    }
end

rule "Give Birth"
    ruleflow-group "evaluate"
    no-loop
  when
    // A dead cell has 3 live neighbors.
    theCell: Cell( liveNeighbors == 3, cellState == CellState.DEAD,
                   phase == Phase.EVALUATE )
  then
    modify( theCell ){
        theCell.setPhase( Phase.BIRTH );
    }
end
```

グリッド内の全 **Cell** オブジェクトが評価されると、この例では **"reset calculate"** ルールを使用して **"calculate"** ルールフローグループのアクティベーションを消去します。次に、ルールグループがアクティベートされると、**"kill"** と **"birth"** のルールを有効にするルールフローに分岐を挿入します。これらのルールにより状態の変更が適用されます。

ルール "reset calculate"、"kill" および "birth"

```
rule "reset calculate"
    ruleflow-group "reset calculate"
  when
  then
    WorkingMemory wm = drools.getWorkingMemory();
    wm.clearRuleFlowGroup( "calculate" );
end

rule "kill"
    ruleflow-group "kill"
    no-loop
  when
    theCell: Cell( phase == Phase.KILL )
  then
    modify( theCell ){
        setCellState( CellState.DEAD ),
        setPhase( Phase.DONE );
    }
end

rule "birth"
    ruleflow-group "birth"
    no-loop
  when
    theCell: Cell( phase == Phase.BIRTH )
  then
    modify( theCell ){
        setCellState( CellState.LIVE ),
        setPhase( Phase.DONE );
    }
end
```

この段階では、複数の **Cell** オブジェクトの状態が **LIVE** または **DEAD** のいずれかに変更されています。この例では、細胞が生存または死亡すると、**"Calculate …"** ルールの **Neighbor** 関係を使用して、周辺の細胞すべてに繰り返し実行し、**liveNeighbor** の数が増減します。数が変更された細胞は、**EVALUATE** フェーズに設定され、ルールフロープロセスの評価段階の理由付けに含められるようにします。

生存数が判断され、全細胞に設定されると、ルールフロープロセスが終了します。ユーザーが最初に **スタート** をクリックすると、その時点でエンジンにより、ルールフローが再起動され、ユーザーが最初に **次の世代** をクリックした場合には、ユーザーは別の世代を要求することができます。

ルール "Calculate …"

```
rule "Calculate Live"
    ruleflow-group "calculate"
    lock-on-active
  when
    theCell: Cell( cellState == CellState.LIVE )
    Neighbor( cell == theCell, $neighbor : neighbor )
  then
    modify( $neighbor ){
        setLiveNeighbors( $neighbor.getLiveNeighbors() + 1 ),
        setPhase( Phase.EVALUATE );
    }
end

rule "Calculate Dead"
    ruleflow-group "calculate"
    lock-on-active
  when
    theCell: Cell( cellState == CellState.DEAD )
    Neighbor( cell == theCell, $neighbor : neighbor )
  then
    modify( $neighbor ){
        setLiveNeighbors( $neighbor.getLiveNeighbors() - 1 ),
        setPhase( Phase.EVALUATE );
    }
end
```

7.9. HOUSE OF DOOM 例のデシジョン (後向き連鎖および再帰)

The House of Doom 例のディシジョンセットでは、デシジョンエンジンが後ろ向き連鎖と再帰を使用して、階層システムで定義した目的やサブゴールに到達するかを例示します。

以下は House of Doom の例の概要です。

- **名前**: **backwardchaining**
- **Main クラス**: (**src/main/java** 内の) **org.drools.examples.backwardchaining.HouseOfDoomMain**
- **モジュール**: **drools-examples**
- **タイプ**: Java アプリケーション

- **ルールファイル**: (**src/main/resources** 内の) **org.drools.examples.backwardchaining.BC-Example.drl**

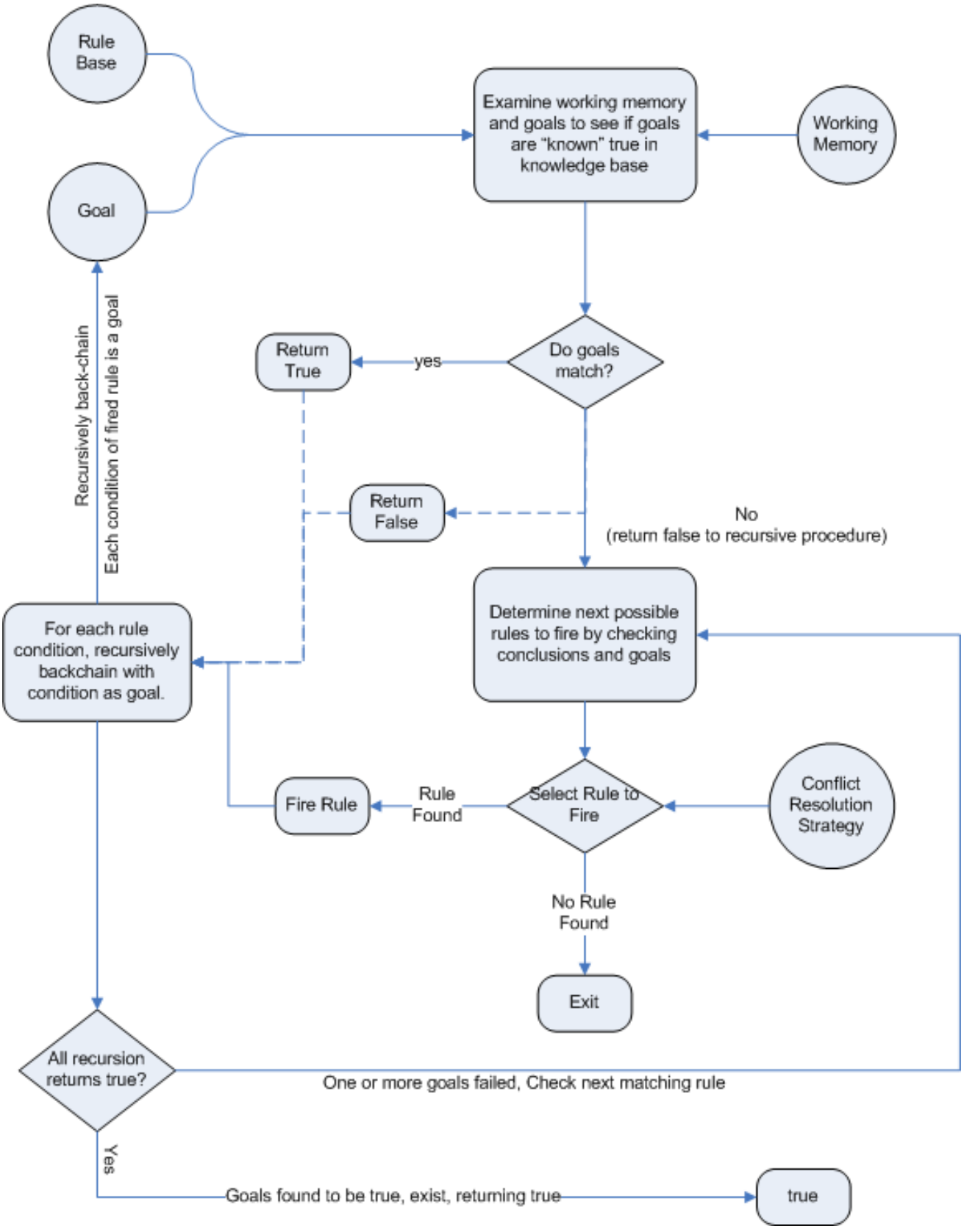
- **目的**: 後向き連鎖と再帰を例示します。

後向き連鎖のルールシステムは、通常再帰を使用して、デシジョンエンジンが満たそうとする結論から開始する目的駆動型のシステムです。システムが結論または目的に到達できない場合には、サブとなる目的、つまり、現在の目的の一部を完了する結論を検索します。システムは、最初の結論が満たされるか、すべてのサブとなる目的が満たされるまで続行されます。

反対に、前向き連鎖のルールシステムは、ディジョンエンジンの作業メモリーにあるファクトで開始して、そのファクトへの変更に反応するデータ駆動型のシステムです。オブジェクトが作業メモリーに挿入されると、その変更の結果として True となってルールの条件が、アジェンダにより実行がスケジュールされます。

以下の図は、後ろ向き連鎖のルールシステムでのロジックフローについて示しています。

図7.23 後ろ向き連鎖のロジックフロー



House of Doom の例は、さまざまなクエリータイプが含まれるルールを使用し、部屋の場所と家の中のアイテムを探し出します。`Location.java` のサンプルクラスには、この例で使用する `item` と `location` 要素が含まれます。`HouseOfDoomMain.java` のサンプルクラスで、家の該当の場所にアイテムまたは部屋を挿入して、ルールを実行します。

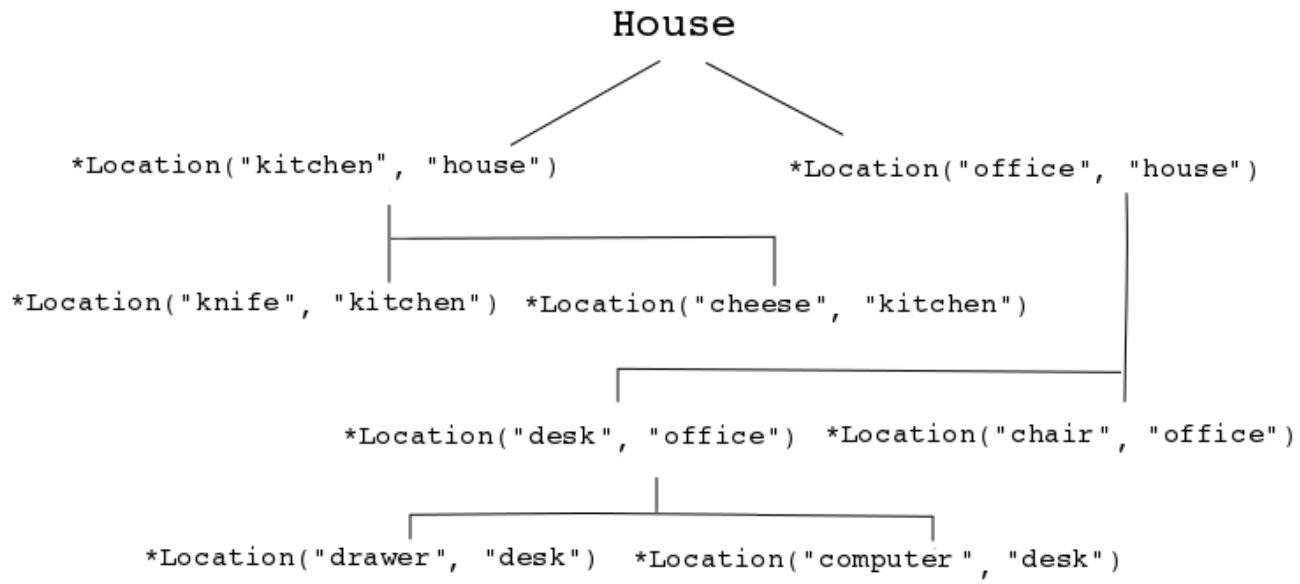
HouseOfDoomMain.java クラスでのアイテムと場所

```
ksession.insert( new Location("Office", "House") );
ksession.insert( new Location("Kitchen", "House") );
ksession.insert( new Location("Knife", "Kitchen") );
ksession.insert( new Location("Cheese", "Kitchen") );
ksession.insert( new Location("Desk", "Office") );
ksession.insert( new Location("Chair", "Office") );
ksession.insert( new Location("Computer", "Desk") );
ksession.insert( new Location("Drawer", "Desk") );
```

ルールの例では、家の構造の中で全アイテムおよび部屋の場所を判断するのに、後向き連鎖と再帰を使用します。

以下の図は、House of Doom の構造と、その構造内のアイテムと部屋を示しています。

図7.24 House of Doom の構造



この例を実行するには、IDE で Java アプリケーションとして **`org.drools.examples.backwardchaining.HouseOfDoomMain`** クラスを実行します。

実行後に、以下の出力が IDE コンソールウィンドウに表示されます。

IDE コンソールでの実行出力

```
go1
Office is in the House
---
go2
Drawer is in the House
---
go3
---
Key is in the Office
---
go4
Chair is in the Office
```

```
Desk is in the Office
Key is in the Office
Computer is in the Office
Drawer is in the Office
---
go5
Chair is in Office
Desk is in Office
Drawer is in Desk
Key is in Drawer
Kitchen is in House
Cheese is in Kitchen
Knife is in Kitchen
Computer is in Desk
Office is in House
Key is in Office
Drawer is in House
Computer is in House
Key is in House
Desk is in House
Chair is in House
Knife is in House
Cheese is in House
Computer is in Office
Drawer is in Office
Key is in Desk
```

この例のルールはすべて実行されて、家の全アイテムの場所を検出し、家の中の全アイテムの場所を検出して、出力でそれぞれの場所を出力します。

再帰クエリーおよび関連のルール
再帰クエリーは、要素間の関係におけるデータ構造階層を使用して繰り返し検索を行います。

House of Doom の例では、**`BC-Example.drl`** ファイルに、この例のルールの大半が使用する **`isContainedIn`** クエリーが含まれており、家のデータ構造を再帰的に評価して、デシジョンエンジンに挿入するデータがないかを確認します。

BC-Example.drl の再帰クエリー

```
query isContainedIn( String x, String y )
  Location( x, y; )
  or
  ( Location( z, y; ) and isContainedIn( x, z; ) )
end
```

`"go"` のルールは、システムに挿入する文字列をすべて出力し、アイテムをどのように導入し、**`"go1"`** ルールが **`isContainedIn`** クエリーを呼び出すかを判断します。

ルール "go" および "go1"

```
rule "go" salience 10
  when
    $s : String( )
  then
    System.out.println( $s );
```

```
end

rule "go1"
  when
    String( this == "go1" )
    isContainedIn("Office", "House"; )
  then
    System.out.println( "Office is in the House" );
end
```

この例は、**"go1"** 文字列をエンジンに挿入して、**"go1"** ルールを有効化し、**House** の場所にある **Office** アイテムを検出します。

文字列の挿入とルールの実行

```
ksession.insert( "go1" );
ksession.fireAllRules();
```

IDE コンソールでの ルール "go1" の出力

```
go1
Office is in the House
```

推移閉包ルール
推移閉包は、階層構造で複数レベル、上層にある親要素に含まれる要素間の関係です。

"go2" ルールは、**Drawer** と **House** の推移閉包の関係を特定します。**Drawer** は、**House** の中の、**Office** の中の、**Desk** の中にあります。

```
rule "go2"
  when
    String( this == "go2" )
    isContainedIn("Drawer", "House"; )
  then
    System.out.println( "Drawer is in the House" );
end
```

この例は、**"go2"** 文字列をエンジンに挿入して、**"go2"** ルールを有効化し、最終的に **House** の場所に含まれる **Drawer** アイテムを検出します。

文字列の挿入とルールの実行

```
ksession.insert( "go2" );
ksession.fireAllRules();
```

IDE コンソールのルール "go2" の出力

```
go2
Drawer is in the House
```

エンジンは、以下のロジックをもとにこの結果を判断します。

1. クエリーは再帰的に、家の中の複数レベルを検索して、**Drawer** と **House** の間の推移閉包を検出します。
2. **Drawer** は **House** に直接含まれないので、**Location(x, y;)** を使用する代わりに、このクエリーは **(z, y;)** の値を使用します。
3. **z** の引数は現在バインドされておらず、値が指定されていないので、引数に含まれるものはすべて返されます。
4. **y** の引数は現在、**House** にバインドされているので、**z** は **Office** と **Kitchen** を返します。
5. クエリーは、**Office** からの情報を収集して、**Drawer** が **Office** に含まれているかを再帰的にチェックします。これらのパラメーターに対して、クエリーの行 **isContainedIn(x, z;)** が呼び出されます。
6. **Office** に直接含まれる **Drawer** が存在しないので、一致するものはいつまりません。
7. **z** のバインドがない場合は、このクエリーでは **Office** 内のデータが返され、**z == Desk** と判断されます。

```
isContainedIn(x==drawer, z==desk)
```

8. **isContainedIn** クエリーは再帰的に 3 回検索し、3 回目に、このクエリーにより **Desk** の中に **Drawer** があることが検出されます。

```
Location(x==drawer, y==desk)
```

9. 最初の場所で上記が一致した後に、このクエリーにより再帰的に構造を上方向に検索し、**Drawer** が **Desk** の中に、**Desk** が **Office** の中に、**Office** が **House** の中にあることを判断します。このように、**Drawer** は **House** の中にあるので、このルールは満たされます。

リアクティブクエリールール

リアクティブクエリーでは、データ構造の階層を検索して、要素間に関係があるかを確認し、構造内の要素が変更されると動的に更新されます。

"go3" ルールは、リアクティブクエリーとして機能し、推移閉包により、新しいアイテム **Key** が **Office** に含まれるかどうかを検出します (**Office** の中の **Key** の中の **Drawer** など)。

ルール "go3"

```
rule "go3"
  when
    String( this == "go3" )
    isContainedIn("Key", "Office"; )
  then
    System.out.println( "Key is in the Office" );
end
```

この例は、**"go3"** 文字列をエンジンに挿入して、**"go3"** ルールを有効化します。最初は、**Key** が家の構造に存在するので、このルールは満たされないため、出力は生成されません。

文字列の挿入とルールの実行

```
ksession.insert( "go3" );
ksession.fireAllRules();
```

IDE コンソールのルール "go3" の出力 (条件を満たさない)

```
go3
```

この例では、**Office** の中にある **Drawer** の場所に、新しいアイテム **Key** を挿入します。この変更で、**"go3"** ルールの推移閉包が満たされ、それに合わせて出力が生成されます。

新規アイテムの場所の挿入とルールの実行

```
ksession.insert( new Location("Key", "Drawer") );
ksession.fireAllRules();
```

IDE コンソールのルール "go3" の出力 (条件を満たす)

```
オフィス内の鍵
```

またこの変更で、クエリーにより、後に続く再帰検索に含まれるよう、この構造に別のレベルが追加されます。

ルールにバインドなしの引数が含まれたクエリー
バインドなしの引数が 1 つ以上あるクエリーでは、クエリーの定義済み (バインドされている) 引数に含まれる未定義 (バインドされていない) アイテムすべてを返します。クエリー内の引数でバインドされているものがない場合には、クエリーはクエリーの範囲内のアイテムをすべて返します。

"go4" ルールは、バインドされている引数を使用して、**Office** 内の特定のアイテムを検索するのではなく、バインドされていない引数 **thing** を使用して、バインドされている引数 **Office** 内の全アイテムを検索します。

ルール "go4"

```
rule "go4"
  when
    String( this == "go4" )
    isContainedIn(thing, "Office"; )
  then
    System.out.println( thing + "is in the Office" );
end
```

この例では **"go4"** 文字列をエンジンに挿入して、**"go4"** ルールをアクティベートし、**Office** の全アイテムを返します。

文字列の挿入とルールの実行

```
ksession.insert( "go4" );
ksession.fireAllRules();
```

IDE コンソールのルール "go4" の出力

```
go4
```

```
Chair is in the Office
Desk is in the Office
Key is in the Office
Computer is in the Office
Drawer is in the Office
```

"go5" ルールは、バインドされていない引数 **thing** と **location** を使用して、**House** の全データ構造の中に含まれる全アイテムとその場所を検索します。

ルール "go5"

```
rule "go5"
  when
    String( this == "go5" )
    isContainedIn(thing, location; )
  then
    System.out.println(thing + " is in " + location );
end
```

この例は **"go5"** 文字列をエンジンに挿入して、**"go5"** ルールをアクティベートし、**House** データ構造に含まれる全アイテムとその場所を返します。

文字列の挿入とルールの実行

```
ksession.insert( "go5" );
ksession.fireAllRules();
```

IDE コンソールのルール "go5" の出力

```
go5
Chair is in Office
Desk is in Office
Drawer is in Desk
Key is in Drawer
Kitchen is in House
Cheese is in Kitchen
Knife is in Kitchen
Computer is in Desk
Office is in House
Key is in Office
Drawer is in House
Computer is in House
Key is in House
Desk is in House
Chair is in House
Knife is in House
Cheese is in House
Computer is in Office
Drawer is in Office
Key is in Desk
```

第8章 次のステップ

- テストシナリオを使用したデシジョンサービスのテスト
- Packaging and deploying a Red Hat Process Automation Manager project

付録A バージョン情報

本ドキュメントの最終更新日: 2019 年 1 月 22 日 (火)